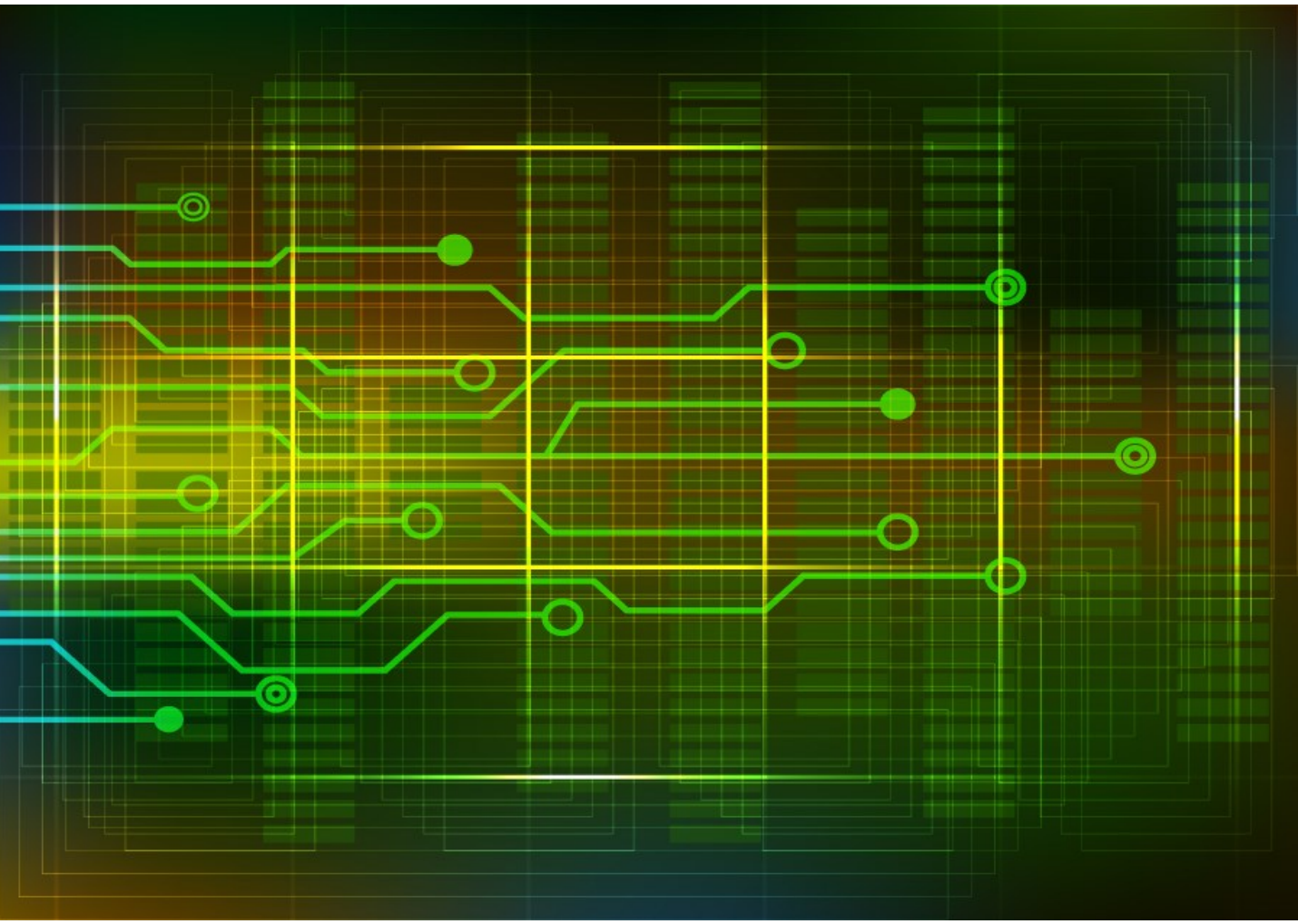
BENOIT BLANCHON
CREATOR OF ARDUINOJSON



Mastering ArduinoJson 7

Efficient JSON serialization for embedded C++

ArduinoJson

Mastering ArduinoJson 7

Published by Benoît Blanchon, Antony, FRANCE.

Logo design by bcendet.

Cover design by Iulia Ghimisli.

Revision date: December 13, 2023

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 changed since Mastering ArduinoJson 6 . . . 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 . . . 8
  - 1.2.5 Why is JSON so popular? . . . 9
  - 1.2.6 The JSON syntax . . . 9
  - 1.2.7 Binary data in JSON . . . 13
  - 1.2.8 Comments in JSON . . . 13
- 1.3 Introduction to ArduinoJson . . . 15
  - 1.3.1 What ArduinoJson is . . . 15
  - 1.3.2 What ArduinoJson is not . . . 15
  - 1.3.3 What makes ArduinoJson different? . . . 16
  - 1.3.4 Does size matter? . . . 18
  - 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 Harvard and von Neumann architectures . . . 31
- 2.3 Stack, heap, and globals . . . 33
  - 2.3.1 Globals . . . 34

2.3.2 Heap . . . 35
2.3.3 Stack . . . 36
2.4 Pointers . . . 38
2.4.1 What is a pointer? . . . 38
2.4.2 Dereferencing a pointer . . . 38
2.4.3 Pointers and arrays . . . 39
2.4.4 Taking the address of a variable . . . 40
2.4.5 Pointer to `class` and `struct` . . . 40
2.4.6 Pointer to constant . . . 41
2.4.7 The null pointer . . . 43
2.4.8 Why use pointers? . . . 44
2.5 Memory management . . . 45
2.5.1 `malloc()` and `free()` . . . 45
2.5.2 `new` and `delete` . . . 45
2.5.3 Smart pointers . . . 46
2.5.4 RAII . . . 48
2.6 References . . . 49
2.6.1 What is a reference? . . . 49
2.6.2 Differences with pointers . . . 49
2.6.3 Reference to constant . . . 50
2.6.4 Rules of references . . . 51
2.6.5 Common problems . . . 51
2.6.6 Usage for references . . . 52
2.7 Strings . . . 53
2.7.1 How are the strings stored? . . . 53
2.7.2 String literals in RAM . . . 53
2.7.3 String literals in Flash . . . 54
2.7.4 Pointer to the “globals” section . . . 55
2.7.5 Mutable string in “globals” . . . 56
2.7.6 A copy in the stack . . . 57
2.7.7 A copy in the heap . . . 58
2.7.8 A word about the `String` class . . . 59
2.7.9 Passing strings to functions . . . 60
2.8 Summary . . . 62

3 Deserialize with ArduinoJson 64
3.1 The example of this chapter . . . 65
3.2 Deserializing an object . . . 66
3.2.1 The JSON document . . . 66
3.2.2 Deserializing the JSON document . . . 66

3.3 Extracting values from an object . . . 68
3.3.1 Extracting values . . . 68
3.3.2 Explicit casts . . . 68
3.3.3 When values are missing . . . 69
3.3.4 Changing the default value . . . 70
3.4 Inspecting an unknown object . . . 71
3.4.1 Getting a reference to the object . . . 71
3.4.2 Enumerating the keys . . . 72
3.4.3 Detecting the type of value . . . 72
3.4.4 Variant types and C++ types . . . 73
3.4.5 Testing if a key exists in an object . . . 73
3.5 Deserializing an array . . . 75
3.5.1 The JSON document . . . 75
3.5.2 Parsing the array . . . 75
3.6 Extracting values from an array . . . 77
3.6.1 Retrieving elements by index . . . 77
3.6.2 Alternative syntaxes . . . 77
3.6.3 When complex values are missing . . . 78
3.7 Inspecting an unknown array . . . 80
3.7.1 Getting a reference to the array . . . 80
3.7.2 Number of elements in an array . . . 80
3.7.3 Iteration . . . 81
3.7.4 Detecting the type of an element . . . 81
3.8 Reading from a stream . . . 83
3.8.1 Reading from a file . . . 83
3.8.2 Reading from an HTTP response . . . 84
3.9 The ArduinoJson Assistant . . . 92
3.9.1 Step 1: Configuration . . . 93
3.9.2 Step 2: JSON . . . 94
3.9.3 Step 3: Program . . . 95
3.10 Summary . . . 96

4 Serializing with ArduinoJson 98
4.1 The example of this chapter . . . 99
4.2 Creating an object . . . 100
4.2.1 The example . . . 100
4.2.2 Creating the JsonDocument . . . 100
4.2.3 Adding members . . . 101
4.2.4 Creating an empty object . . . 101
4.2.5 Replacing and removing members . . . 102

4.3 Creating an array . . . 103
4.3.1 The example . . . 103
4.3.2 Adding elements . . . 103
4.3.3 Adding nested objects . . . 104
4.3.4 Creating an empty array . . . 105
4.3.5 Replacing and removing elements . . . 105
4.4 Writing to memory . . . 106
4.4.1 Minified JSON . . . 106
4.4.2 Specifying (or not) the buffer size . . . 106
4.4.3 Prettified JSON . . . 107
4.4.4 Measuring the length . . . 108
4.4.5 Writing to a `String` . . . 109
4.4.6 Casting a `JsonVariant` to a `String` . . . 109
4.5 Writing to a stream . . . 110
4.5.1 What's an output stream? . . . 110
4.5.2 Writing to the serial port . . . 111
4.5.3 Writing to a file . . . 112
4.5.4 Writing to a TCP connection . . . 113
4.6 Duplication of strings . . . 118
4.6.1 An example . . . 118
4.6.2 Keys and values . . . 119
4.6.3 Copy only occurs when adding values . . . 119
4.7 Inserting special values . . . 120
4.7.1 Adding `null` . . . 120
4.7.2 Adding pre-formatted JSON . . . 120
4.8 The ArduinoJson Assistant . . . 122
4.8.1 Step 1: Configuration . . . 122
4.8.2 Step 2: JSON . . . 123
4.8.3 Step 3: Program . . . 123
4.9 Summary . . . 125

5 Advanced Techniques 127
5.1 Introduction . . . 128
5.2 Filtering the input . . . 129
5.3 Deserializing in chunks . . . 134
5.4 JSON streaming . . . 139
5.5 Using external RAM . . . 142
5.6 Logging . . . 145
5.7 Buffering . . . 148
5.8 Custom readers and writers . . . 151

5.9 Custom converters . . . 156
5.10 MessagePack . . . 162
5.11 ArduinoJson Assistant's Tweaks . . . 165
5.11.1 Floating-point storage . . . 166
5.11.2 Integer storage . . . 166
5.11.3 String deduplication . . . 167
5.11.4 `const char*` strings . . . 168
5.12 Summary . . . 169

6 Inside ArduinoJson **171**
6.1 Introduction . . . 172
6.2 Variants . . . 173
6.3 Integers . . . 175
6.4 String . . . 177
6.4.1 String nodes . . . 177
6.4.2 String adapters . . . 178
6.4.3 Variable-length arrays . . . 178
6.5 Arrays and objects . . . 180
6.6 Document tree . . . 182
6.7 Slot pool . . . 184
6.8 The resource manager . . . 186
6.9 Smart pointers . . . 187
6.9.1 `JsonVariant` . . . 187
6.9.2 Unbound `JsonVariant` . . . 187
6.9.3 `JsonVariantConst` . . . 188
6.9.4 `JsonArray` and `JsonObject` . . . 189
6.10 Comparison operators . . . 190
6.11 Converters . . . 191
6.12 Proxies . . . 193
6.13 Deserializers . . . 195
6.13.1 Reading from various types . . . 195
6.13.2 Reading one character at a time . . . 196
6.13.3 Nesting limit . . . 196
6.13.4 Escape sequences . . . 198
6.13.5 Filtering . . . 199
6.13.6 String-to-float conversion . . . 199
6.14 Serializers . . . 200
6.14.1 Writing to various types . . . 200
6.14.2 Float-to-string conversion . . . 201
6.15 Namespaces . . . 202

6.16 Aggregated header . . . 203
6.17 Summary . . . 204

7 Troubleshooting 206
7.1 Introduction . . . 207
7.2 Program crashes . . . 208
7.2.1 Undefined Behaviors . . . 208
7.2.2 A bug in ArduinoJson? . . . 208
7.2.3 Null string . . . 209
7.2.4 Use after free . . . 209
7.2.5 Return of stack variable address . . . 211
7.2.6 Buffer overflow . . . 212
7.2.7 Stack overflow . . . 213
7.2.8 How to diagnose these bugs . . . 214
7.2.9 How to prevent these bugs? . . . 216
7.3 Deserialization issues . . . 219
7.3.1 `EmptyInput` . . . 219
7.3.2 `IncompleteInput` . . . 220
7.3.3 `InvalidInput` . . . 222
7.3.4 `NoMemory` . . . 226
7.3.5 `TooDeep` . . . 226
7.4 Serialization issues . . . 228
7.4.1 The JSON document is incomplete . . . 228
7.4.2 The JSON document contains garbage . . . 228
7.4.3 The serialization is slow . . . 229
7.5 Common error messages . . . 231
7.5.1 Invalid conversion from `const char*` to `char*` . . . 231
7.5.2 Invalid conversion from `const char*` to `int` . . . 231
7.5.3 No match for `operator[]` . . . 232
7.5.4 Ambiguous overload for `operator=` . . . 233
7.5.5 Call of overloaded function is ambiguous . . . 234
7.6 Asking for help . . . 235
7.7 Summary . . . 237

8 Case Studies 238
8.1 JSON Configuration File . . . 239
8.1.1 Presentation . . . 239
8.1.2 The JSON document . . . 239
8.1.3 The configuration class . . . 240
8.1.4 Converters . . . 241

8.1.5 Saving the configuration to a file . . . 244
8.1.6 Reading the configuration from a file . . . 245
8.1.7 Conclusion . . . 245
8.2 OpenWeatherMap on MKR1000 . . . 247
8.2.1 Presentation . . . 247
8.2.2 OpenWeatherMap's API . . . 247
8.2.3 The JSON response . . . 248
8.2.4 Reducing the size of the document . . . 250
8.2.5 The filter document . . . 252
8.2.6 The code . . . 253
8.2.7 Summary . . . 254
8.3 Reddit on ESP8266 . . . 255
8.3.1 Presentation . . . 255
8.3.2 Reddit's API . . . 256
8.3.3 The response . . . 257
8.3.4 The main loop . . . 258
8.3.5 Sending the request . . . 259
8.3.6 Assembling the puzzle . . . 259
8.3.7 Summary . . . 260
8.4 RESTful client . . . 262
8.4.1 Presentation . . . 262
8.4.2 JSON-RPC Request . . . 263
8.4.3 JSON-RPC Response . . . 263
8.4.4 A reusable RESTful client . . . 264
8.4.5 Sending notification to Kodi . . . 268
8.4.6 Reading Kodi's version . . . 270
8.4.7 Summary . . . 272
8.5 Recursive analyzer . . . 273
8.5.1 Presentation . . . 273
8.5.2 Reading from the serial port . . . 274
8.5.3 Flushing after an error . . . 275
8.5.4 Testing the type of a `JsonVariant` . . . 275
8.5.5 Printing values . . . 277
8.5.6 Summary . . . 279

9 Conclusion 280

Index 281

Chapter 1
Introduction

> *We see a lot of feature-driven product design in which the cost of features is not properly accounted. Features can have a negative value to consumers because they make the products more difficult to understand and use. We are finding that people like products that just work. It turns out that designs that just work are much harder to produce that designs that assemble long lists of features.*
>
> **– 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. This book is a complete guide to ArduinoJson 7. It covers all the library's features, from the basics to the most advanced techniques. It also explains how the library works under the hood.

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 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 7.0**; you can download the code samples from `files.arduinojson.org/v7/maj.zip`

1.1.3 What changed since Mastering ArduinoJson 6

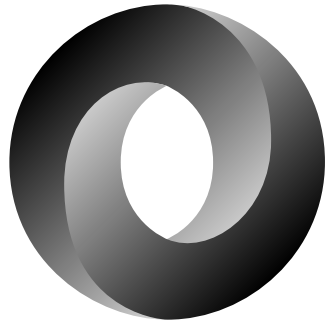
For this new revision, I preserved most of the content and updated it for ArduinoJson 7. In addition:

- I removed all workarounds for C++03, since ArduinoJson 7 requires C++11.
- In the "Advanced Techniques" chapter:
  - I removed the sections "Automatic capacity" and "Fixing memory leaks" as they are no longer relevant.
  - I added a section about the "Tweaks" of the ArduinoJson Assistant.
- I rewrote the "Inside ArduinoJson" chapter because of the significant change in the library and because I didn't like this chapter.
- In the "Case studies" chapter:
  - I replaced SPIFFS with LittleFS because the former is now deprecated.
  - I rewrote the Kodi case study to include a reusable REST client class.
- As with each new book revision, I updated the list of noteworthy web APIs. Below is the list of services I removed because they are no longer available, proving that you must be cautious when choosing a service provider, or your project might stop working without notice.
  - DarkSky: acquired by Apple
  - Google IoT Core: retired by Google in 2023
  - Xively: purcharsed by Google
  - ImperiHome: bankrupt
  - automate.io: acqui-hired by Notion
  - Integromat: acquired by make.com who discontinued the service

The book is now significantly shorter than the previous version because ArduinoJson 7 is much easier to use.

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 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:

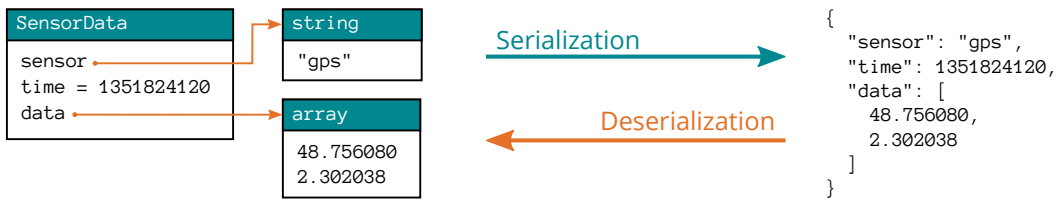
```
{
  "sensor": "gps",
  "time": 1351824120,
  "data": [
    48.756080,
    2.302038
  ]
}
```

One says that the first JSON document is "minified" and that 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 that 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:

- 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)
- Dictionaries
  - Oxford Dictionaries (developer.oxforddictionaries.com)
  - Wordnik (developer.wordnik.com)
- Finance:
  - Binance (binance.com)
  - Coinranking (coinranking.com)
- Home automation
  - Domoticz (domoticz.com)
  - Home Assistant (home-assistant.io)
  - Jeedom (jeedom.com)
  - openHAB (openhab.org)
- Internet of Things (IoT)
  - Adafruit IO (io.adafruit.com), we'll see an example in the fourth chapter
  - Arduino IoT Cloud (arduino.cc/reference/en/iot/api/)
  - Blynk (blynk.io)
  - Dweet (dweet.io)
  - ThingsBoard (thingsboard.io)
  - ThingSpeak (thingspeak.com)
  - Temboo (temboo.com)
- IP geolocation
  - ip-api (ip-api.com)
  - ipstack (ipstack.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)
- News:
  - NewsAPI (newsapi.org)
  - The Guardian (open-platform.theguardian.com)
- Quotes
  - Advice Slip (api.adviceslip.com)
  - Quotable (api.quotable.io)
- Social networks
  - Facebook (developers.facebook.com)
  - Instagram (developers.facebook.com)
  - Twitter (developer.twitter.com)
- Task automation
  - IFTTT (ifttt.com)
  - Microsoft Power Automate (flow.microsoft.com)
  - Workato (workato.com)
  - Zapier (zapier.com)
- Time
  - Timezonedb (timezonedb.com)
  - WorldTimeAPI (worldtimeapi.org)
- Translation
  - Google Translate (cloud.google.com)
  - Microsoft Translator (azure.microsoft.com)
- Weather forecast
  - AccuWeather (accuweather.com)
  - OpenWeatherMap (openweathermap.org), we'll see an example in the case studies

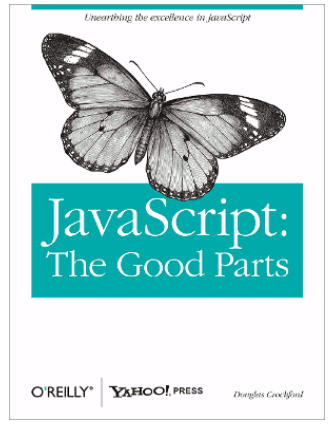
This list is not exhaustive; you can find many more examples. If you wonder whether a specific web service has a JSON API, search for the following terms in the developer documentation: "API," "HTTP API," "REST API," or "webhook."

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.

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. 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?

The success of JSON can be largely attributed to the frustration caused by XML. 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.

XML is very powerful, but it is overkill for most projects. The syntax is very verbose because you need to repeat the opening and closing tags. Moreover, the angle brackets make XML documents hard to read for humans.

But it's not just humans who struggle with XML; machines do, too. Serializing and deserializing XML is complicated because tags have children *and* attributes (something that doesn't exist in JSON). Special characters must be encoded in a nontrivial way (for example `>` becomes `>`), and the `CDATA` sections must be handled entirely differently.

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.

1.2.6 The JSON syntax

You can find the format specification on `json.org`; we'll only see 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

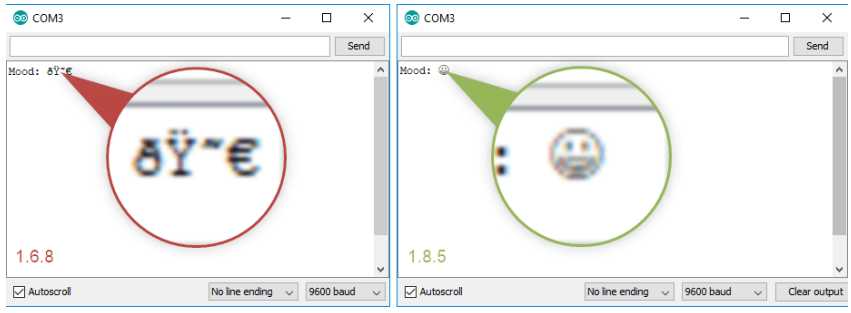
In JSON, the strings are surrounded by double quotes. JSON is more restrictive than JavaScript, which also supports single quotes. ArduinoJson supports both.

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 the UTF-8 encoding, so UTF-16 escape sequences are unnecessary. By default, ArduinoJson decodes Unicode UTF-16 escape sequence into UTF-8 characters, but it doesn't perform the reverse operation.

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 be surrounded by double quotes. JavaScript also supports single quotes too and even allows single-word keys without any quote. 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, and the most important is its inability to transmit raw (meaning unmodified) binary data. Indeed, to send binary data in JSON, you must either use an array of integers 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 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 convenient for configuration files, so many 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 that serializes and deserializes JSON documents. It is designed to work in embedded environments, i.e., on devices with very limited power.

Because it is open-source and has a 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 in 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 what 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 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 the JSON model offers, 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.

The ArduinoJson parser 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 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 library includes several optimizations to reduce memory usage. For example, it employs memory pools to minimize heap fragmentation. It deduplicates strings so that only one copy of each string is kept in memory. It doesn't copy the string literals to avoid

duplicating a string that is already in RAM. We'll talk about all these optimizations in Inside ArduinoJson.

ArduinoJson is a header-only library, meaning all the library code 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! You can even use the library with web compilers like `wandbox.org`; go to the ArduinoJson website and 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 so you can use it in any C++ project. For example, you can run unit tests and debug your program 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.

Sometimes, you don't need the entire JSON document but only a few values. In this case, you can use ArduinoJson's filtering feature to keep only the values you are interested in, saving a lot of memory. We'll see all the details 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	`Print`	`std::ostream`
Input stream	`Stream`	`std::istream`
String in RAM	`String`	`std::string`
String in Flash	`__FlashStringHelper`	

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.

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 matter?

Let's take a concrete example to show how vital program size and memory usage are. 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 17460 bytes (54%) of program storage space. Maximum is 32256
↪  bytes.
Global variables use 966 bytes (47%) of dynamic memory, leaving 1082 bytes
↪  for local variables. Maximum is 2048 bytes.
```

Yep. That is right. The skeleton already takes 54% of the Flash memory and 47% 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 22120 bytes (68%) of program storage space. Maximum is 32256
↪  bytes.
Global variables use 1008 bytes (49%) of dynamic memory, leaving 1040 bytes
↪  for local variables. Maximum is 2048 bytes.
```

ArduinoJson added only 4660 bytes of Flash and 42 bytes of RAM to the program, which is very small considering all the features that it supports. The library represents only 21% of the program's size but enables a wide range of applications.

Still too big?

If program space is very limited, you can downgrade to ArduinoJson 6. Its fixed allocation strategy allows it to be 38% smaller in this case.

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 lengths, you need a library. Here are four alternatives for Arduino.

Arduino_JSON

Arduino_JSON is the "official" JSON library provided by the Arduino organization. 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 with 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, making 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 made 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 bigger than the device's RAM. 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.

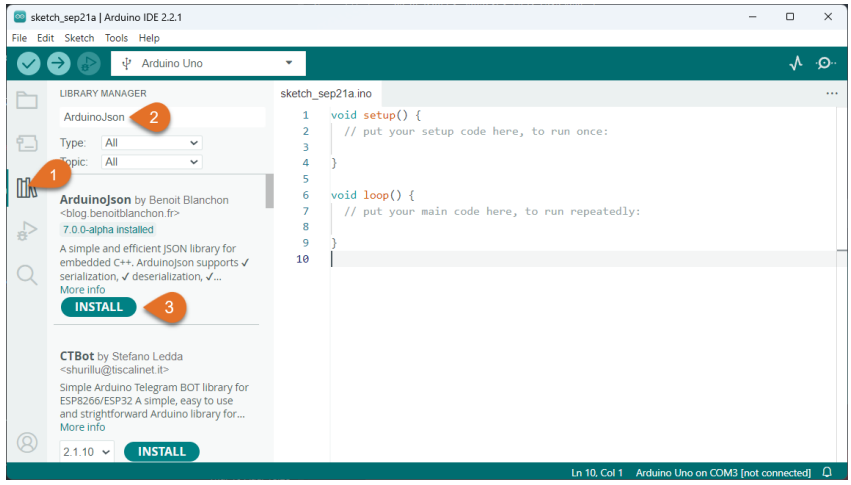
I often recommend this library when ArduinoJson is not suitable.

1.3.6 How to install ArduinoJson

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

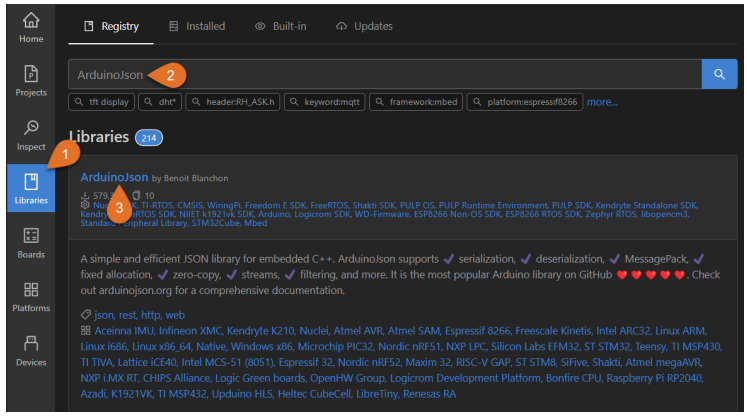
Option 1: using the Arduino Library Manager

If you use the Arduino IDE, you can install ArduinoJson directly from the IDE, thanks to the "Library Manager."

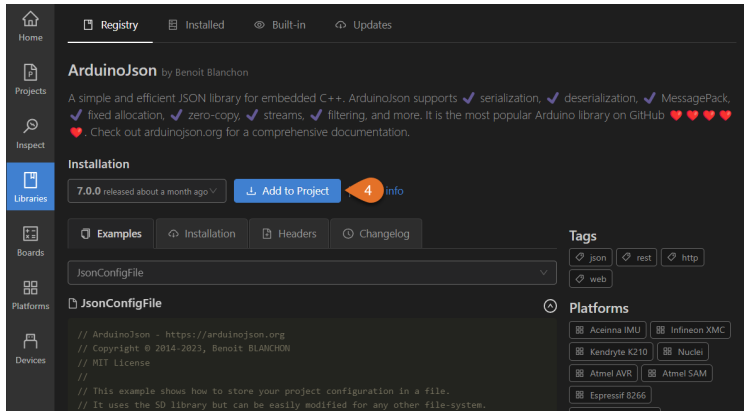


Option 2: using the PlatformIO IDE

Like the Arduino IDE, the PlatformIO IDE offers a simple way to install libraries. Click on the PlatformIO icon on the left, then click "Libraries," and type "ArduinoJson," in the search box.



Click on the library's name and then click on "Add to Project."



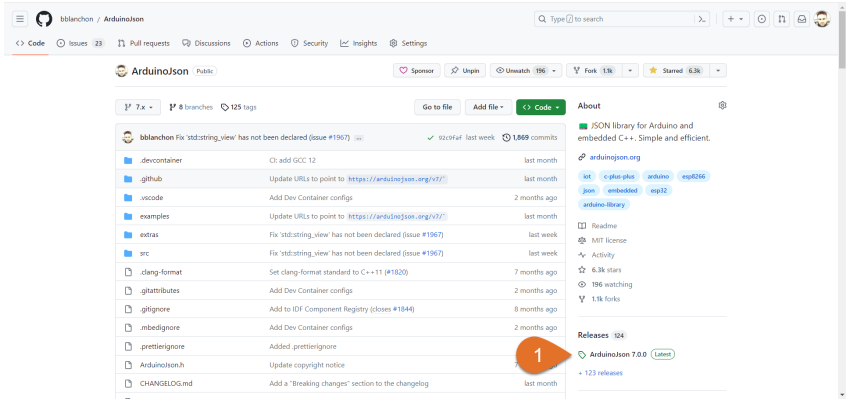
Alternatively, you can install ArduinoJson through the PlatformIO CLI, like so:

```
pio lib install ArduinoJson
```

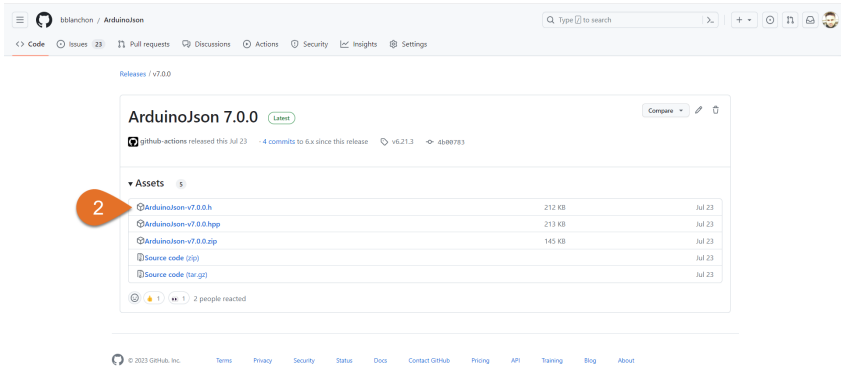
Option 3: using the "single header" distribution

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.

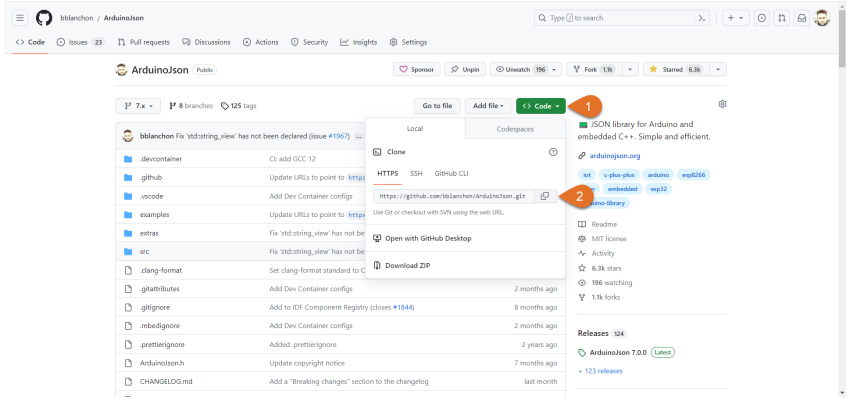
Save this file in your project folder alongside your source code.

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

Option 4: 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.

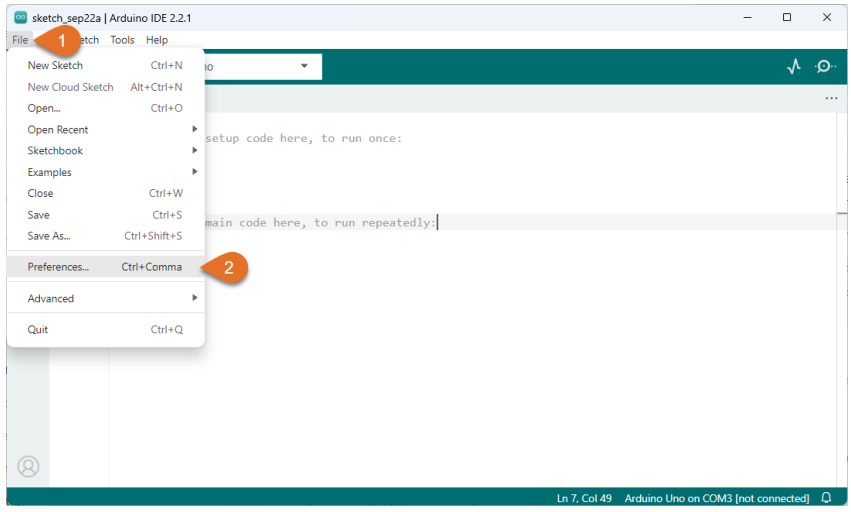
To find the URL of the ArduinoJson repository, go to GitHub and click on "Clone or download."



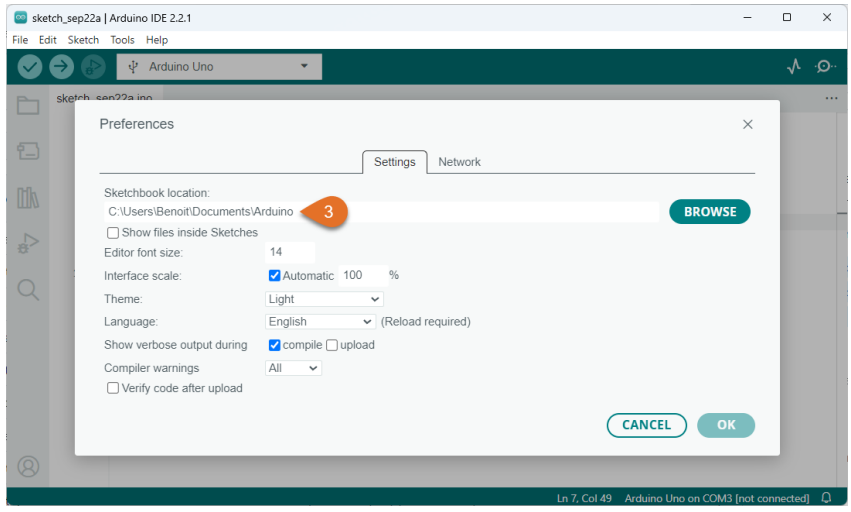
Then, copy the URL to the clipboard and use your favorite Git client to clone the repository in Arduino's libraries folder, which is:

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

The "Arduino Sketchbook folder" is configured in the Arduino IDE; it's in the "Preferences" window, accessible via the "File" menu.



The setting is named "Sketchbook location."



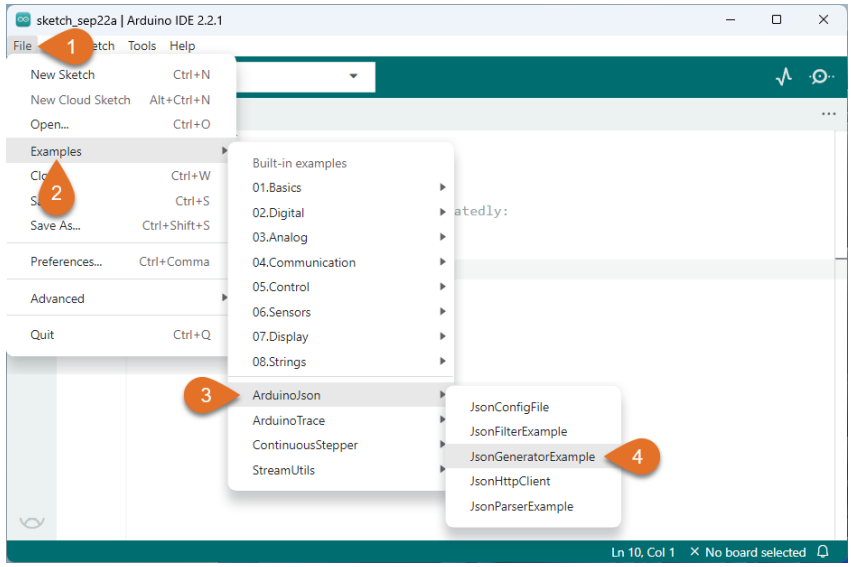
For example, on my computer, the clone folder would be:

```
C:\Users\benoit\Documents\Arduino\libraries\ArduinoJson
```

Finally, I'll need to restart the Arduino IDE to make it aware of the new library.

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/v7/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 for transmitting binary data.
- ArduinoJson works everywhere, not just on Arduino.
- ArduinoJson is a serialization library, not a container library.
- ArduinoJson can filter the input to save memory.
- 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 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 many beginner 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 those 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()`

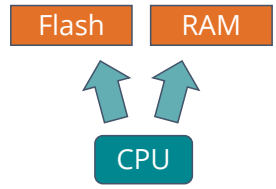
3. How to write simple C / C++ programs:
   - `#include`
   - `class` / `struct`
   - `void`, `int`, `float`, `String`
   - functions

2.2 Harvard and von Neumann architectures

Microcontrollers use two kinds of memory: Flash memory stores the program, and RAM stores the variables. The Flash memory is non-volatile (it preserves the values when you power off the chip), whereas the RAM is volatile (it loses all information when the power is off). The Flash memory is bigger but slower than the RAM. In addition to the program, the Flash memory sometimes stores other data, such as files, but it's not relevant for this chapter.

There are two ways to present these memories to the CPU: they can be seen as two devices or merged into one memory space. How the CPU accesses the two memories depends on the microcontroller architecture. The two most common kinds are the "Harvard" and "von Neumann" architectures.

The **Harvard architecture** uses different address spaces for RAM and Flash. Because the two spaces are unrelated, the same address can refer to one or the other. This ambiguity forces the CPU to use different instructions for Flash and RAM, which simplifies the hardware but complexifies the software.

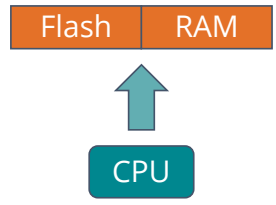


With Harvard microcontrollers, constants are copied to RAM by default, but we can use special attributes to avoid this and save some space. We'll talk about that in the section dedicated to strings.

The following microcontrollers use the Harvard architecture:

- AVR (Uno R3, Leonardo, Nano, Mega…)
- ESP8266

The **von Neumann architecture** uses the same address space for RAM and Flash. A range of addresses is reserved for the Flash memory and another for RAM, so a memory address is unambiguous. This architecture complexifies the hardware but simplifies the software because the CPU can use the same instructions to deal with both memories.



With von Neumann microcontrollers, the constants don't need to be copied to RAM, so they don't suffer from the problem mentioned above.

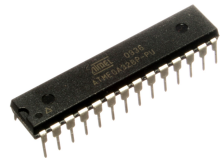
The following microcontrollers use the von Neumann architecture:

- ESP32
- megaAVR (Uno WiFi Rev2, Nano Every)
- SAMD (Zero, MKR, Nano 33...)
- STM32 (Nucleo, Disco, Maple Mini, LoRa...)
- nRF51 (micro:bit), nRF52
- RA4M1 (Uno R4)

2.3 Stack, heap, and globals

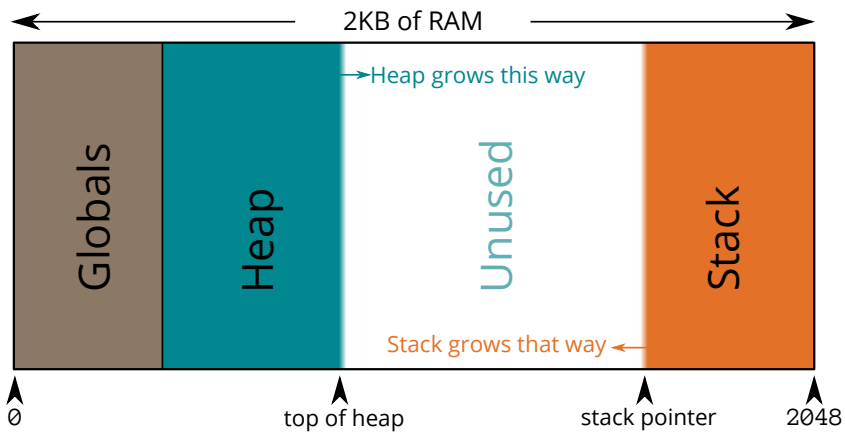
In this section, we'll talk about the RAM of the microcontroller and how the program uses it. The goal here is not to be 100% accurate but to give you the right mental model to understand how C++ deals with memory management.

We'll use the microcontroller Atmel ATmega328 as an example. This chip powers many original Arduino boards (UNO, Duemilanove, etc.) and 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 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.

2.3.1 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 program's execution. All the variables 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 f() {
  return i;
}
```

In this case, the variable `i` occupies 2 bytes in the "globals" area. These two bytes are always present in memory, even when the program is not using them.

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

Harvard architecture only. 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:

```
// Harvard architecture only (AVR and ESP8266)
// the following string occupies 12 bytes in the "globals" area
const char* myString = "hello world";
```

To prevent string literals from eating the whole RAM, you can ask the compiler to keep 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 use them

because they are not regular strings anymore. We will see that later when we talk about strings.

Von Neumann architecture doesn't suffer from this problem because the constants (including the string literals) are kept in Flash memory and don't need to be copied to RAM.

2.3.2 Heap

The "heap" contains the variables that are dynamically allocated. Unlike the "globals" area, its size varies during the program's execution. 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.

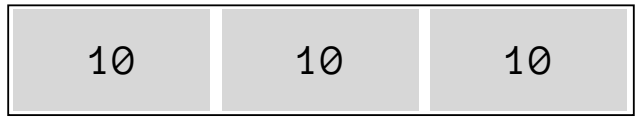
Here is a program that allocates 42 bytes in the heap and then releases them:

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

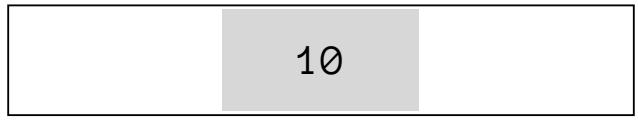
I insist on the fact that managing the heap is the program's role and not the programmer's role. Indeed, it is a common misunderstanding that, in C++, memory must be managed manually by the programmer; in reality, the language does that for us, as we'll see in the section dedicated to memory management.

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 reduce RAM usage, but it also reduces the size of the program because the memory management functions can be removed.

ArduinoJson 6 allows you to work without a heap, but not ArduinoJson 7.

2.3.3 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 program's execution. 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 invocation site when the function returns.

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

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

In this case, the variable `i` occupies 2 bytes in the "stack" area. These two bytes are only present in memory when the program executes the function `f()`.

Stack size

While this is not true 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 stack memory gets overwritten, corrupting the return addresses, which makes the program jump to an incorrect location. This phenomenon is called a "stack overflow" and is the cause of many crashes.

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.

2.4 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.

2.4.1 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 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.

In reality, the pointer's value 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`. But, 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.

2.4.2 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
0x100	42
0x101	43
0x102	44
...	...

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 must specify the type of value pointed by the pointer; it's also the type of value 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);
```

2.4.3 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. They are equivalent most of the time; you can use an array as a pointer and a pointer as an array. In fact, function parameters declared as arrays are actually pointers: `void f(int a[])` is equivalent to `void f(int* a)`.

2.4.4 Taking the address of a variable

Up till now, we hard-coded the pointer's value, 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 uses 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 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`.

2.4.5 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 who 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, in which case you must 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;
```

2.4.6 Pointer to constant

When you lend a disk to a friend, you hope she will return it 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 receives a pointer as a parameter. 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()`, which 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 ensure that all functions 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.

2.4.7 The null pointer

Zero is a special value representing an empty pointer, just as you'd use `null` in Java, JavaScript, and C#, `None` in Python, or `nil` in Ruby or Swift.

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 :-(
}
```

Don't use `NULL`

`NULL` is a heritage from the C programming language that messes with the type system. Don't use `NULL` in C++ programs; use `nullptr` instead.

For more information, see my article Why C++ programmers don't use NULL?

2.4.8 Why use pointers?

C programmers use pointers for the following tasks:

1. To access 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 use pointers too, but they prefer references when possible. We'll see that in the dedicated section.

2.5 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.

2.5.1 `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 how C programmers manage their memory, but C++ programmers avoid `malloc()` and `free()` because they don't call constructors or destructors.

2.5.2 `new` and delete

The C++ versions of `malloc()` and `free()` are the `new` and `delete` operators. 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, JavaScript, 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 challenging 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, JavaScript, 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.

2.5.3 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:

```cpp
// Same structure as before
struct MyStruct {
  MyStruct() {} // constructor
  ~MyStruct() {} // destructor
  void 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
  ~MyStructPtr() {
    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 syntax 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 similar to Java, JavaScript, or C#.

Unfortunately, these classes are usually not available on Arduino; only a few cores (notably the ones for ESP8266 and ESP32) have them.

2.5.4 RAII

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. In this context, a resource can be either a memory block, a file, a mutex, etc.

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.

I wish I could also give you an example where the resource is a file, but unfortunately, none of the libraries I checked (SD, LittleSF, SPIFFS) implement RAII in their `File` class, which is clearly a mistake.

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.

2.6 References

2.6.1 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.

2.6.2 Differences with pointers

We could rewrite the example above 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.

2.6.3 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 creating 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.

2.6.4 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.

2.6.5 Common problems

Despite all these properties, the references share some vulnerabies with 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 bulletproof.

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.

2.6.6 Usage for references

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

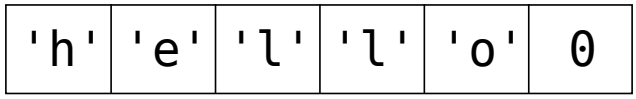
1. Passing parameters to function when copying is expensive.
2. Keeping a dependency on an object that we do not own.

2.7 Strings

2.7.1 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 bytes:



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.

2.7.2 String literals in RAM

There are many ways to declare a string in C++, but for us, the most important ones are the following three:

```
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 Flash memory. As we saw, microcontrollers based on the Harvard architecture (AVR and ESP8266) copy these 6 bytes to RAM when the program boots. These bytes end up in the "globals" area and limit the remaining RAM for the rest of the program; we'll see how to avoid that in the next section.

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.

2.7.3 String literals in Flash

Harvard architecture only

This section is only relevant for microcontrollers based on the Harvard architecture (AVR and ESP8266). Microcontrollers based on the von Neumann architecture (ESP32, ARM…) won't benefit from these techniques.

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 avoid the copy in RAM:

```
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 data space.

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

Fortunately, many functions in Arduino (notably in `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 use a pointer `const __FlashStringHelper*` instead of a regular pointer `const char*`.

For example, to print a Flash string to the serial port, we must write:

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

Failing to use the appropriate pointer type is usually sanctioned by an immediate crash. You've been warned!

Admittedly, this syntax is not very convenient. Fortunately, 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"));
```

There is a catch, though. 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.

Also, you must be aware that most functions need to copy the Flash string to RAM before using it. For example, `String(F("hello"))` will copy `"hello"` to the heap. So, don't abuse Flash strings, or you might 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.

2.7.4 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" area. 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" area 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 to alter the string depends on the platform. On an ATmega328, it will work because there is no memory protection. However, on other platforms, such as 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 to the same memory location
  // proving that there is only one copy of "hello"
} else {
  // this will never happen, even with optimizations disabled
}
```

We'll see in a moment why `s1 == s2` may not do what you think.

2.7.5 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.

This syntax defeats the string interning, as you can see with this snippet:

```
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
}
```

String objects, like the `String` or `JsonVariant`, call this function internally when you use the `==` operator.

2.7.6 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 fills them with the string's characters. In addition to the stack, a copy of the string `"hello"` is

still present in the "globals" area. On Harvard architecture, it means that two copies of the string are in memory.

Code smell

This syntax causes the same string to be present twice in RAM on Harvard architecture. 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 `chars` in the stack, don't initialize it:

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

2.7.7 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 it in the heap. The third syntax presented was:

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

Thanks to the 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, passing a pointer to the contructor. The constructor makes a dynamic memory allocation (using `malloc()`) and copies the characters.

Code smell

This syntax causes the same string to be present twice in RAM on Harvard architecture.

From Flash to heap

To avoid the useless copy on Harvard architecture, you can combine the `F()` macro with the `String` constructor, like so:

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

2.7.8 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;

- String s = String("Answer is ") + answer;
+ 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, PSTR("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. Prefer fixed-size strings to variable-size strings because they are faster, smaller, more reliable and don't fragment the heap.

2.7.9 Passing 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 "hello world" string 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 function's parameter 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 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 object's address. 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 C++ always pass them by address, so the string isn't copied either.

As we saw, arrays and pointers are often interchangeable. Here is another example:

```
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 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
```

2.8 Summary

That's the end of this express 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 created a blog called "C++ for Arduino," where I shared some C++ recipes for Arduino programmers. It's not very active, but still contains some valuable things. Please have a look at `cpp4arduino.com`.

Here are the key points to remember from this chapter:

- The two kinds of memory:
  - Flash is non-volatile and stores the program.
  - RAM is volatile and stores the data.
- The two computer architectures:
  - Harvard (AVR, ESP8266) has two address spaces.
  - Von Neumann (ARM, ESP32) has one address space.
- The three areas of RAM:
  - The "globals" area contains the global variables. It also contains the strings on Harvard architectures.
  - 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.
  - Use `nullptr` instead of `NULL` to declare a null pointer.
  - To dereference a pointer, we 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 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 they call constructors and destructors.
  - You don't have to call `free()` or `delete` explicitly; instead, you should use a smart pointer to do it for you.
  - Smart-pointer is an instance of RAII, the most fundamental idiom in C++.
  - Heap fragmentation reduces the perceived 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; you should use `char` arrays instead.

If you're looking for a good book as your next step to learning C++, I recommend A Tour of C++ by Bjarne Stroustrup, the creator of the language. As the name suggests, this book takes you on a tour to discover all the important features of C++ but doesn't go into the details. It's perfect for experienced programmers who want a quick overview of C++. Then, if you are looking for a more in-depth approach, I recommend Programming: Principles and Practice Using C++ from the same author.

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

Chapter 3

Deserialize with ArduinoJson

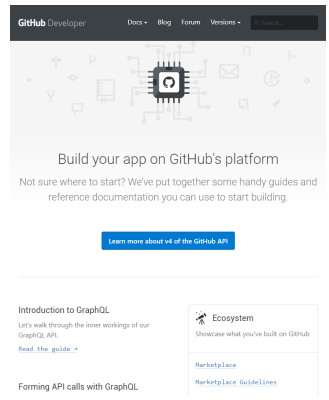
It is not the language that makes programs appear simple. It is the programmer that makes the language appear simple!

– 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.



We could do many things 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 get all the information we need with only one query. It also returns much smaller responses than v3, which is appreciable for embedded software.

To run this 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 we'll finally be able to interact with GitHub by the end of the chapter.

3.2 Deserializing an object

We'll begin this tutorial with the simplest situation: a JSON document in memory. Later, we'll see how to read a JSON document from a file and then an HTTP response.

3.2.1 The JSON document

Our example is the repository information for ArduinoJson:

```
{
  "name": "ArduinoJson",
  "stargazers": {
    "totalCount": 6287
  },
  "issues": {
    "totalCount": 22
  }
}
```

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

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

```
const char* input = "{\"name\":\"ArduinoJson\",\"stargazers\":{"
                    "\"totalCount\":6287},\"issues\":{\"totalCount\":22}}"
```

3.2.2 Deserializing the JSON document

To parse this JSON document, we need a `JsonDocument` to store the result:

```
JsonDocument doc;
```

`JsonDocument` is the cornerstone of ArduinoJson. It's a data structure that represents a JSON document in memory.

Now that we have the input and the `JsonDocument`, 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::EmptyInput`: the input was empty or contained only spaces.
- `DeserializationError::IncompleteInput`: the input was valid but ended prematurely.
- `DeserializationError::InvalidInput`: the input was not a valid JSON document.
- `DeserializationError::NoMemory`: the `JsonDocument` was too small.
- `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. It also has an `f_str()` member that returns a Flash string, saving some space on Harvard architectures like ESP8266.

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

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

Some programmers avoid implicit casts because they mess with overload resolution, template parameter type deduction, and 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 the expression on the right. It is the equivalent of `var` in C# and Java.

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

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

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 next. I use the explicit version only to solve ambiguities.

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
`const char*`	`nullptr`
`float, double`	`0.0`
`int, long…`	`0`
`String`	`""`
`JsonArray`	a null object
`JsonObject`	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

Exceptions are an excellent C++ feature, but they produce large executables, so most embedded software is built with the `-fno-exceptions` flag, which disables exceptions. For this reason, ArduinoJson never throws exceptions.

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 for specifying default configuration values, like in the snippet above, but it is even more helpful 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];
strlcpy(hostname, config["hostname"] | "arduinojson.org", 32);
```

`strlcpy()`, 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)
  strlcpy(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 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 generic container: it can hold an object, an array, or any other value allowed by JSON. Because it's generic, `JsonDocument` only offers methods that apply unambiguously to objects, arrays, and other supported types.

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 object's member because it doesn't know, at compile-time, whether it should behave as an object or 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 alter 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. However, 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-value pair is represented by the `JsonPair` class.

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()`.

3.4.3 Detecting the type of value

Like `JsonObject`, `JsonVariant` is a reference to a value stored in the `JsonDocument`. However, it is not limited to objects and can refer to any JSON value: string, integer, array, object, etc. 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<const 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<const 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, and 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	`bool`
Integer	`char`, `int`, `long`, `long long`, `short` (all `signed` and `unsigned`)
Float	`float`, `double`
String	`const char*`, `String`, `std::string`, `std::string_view`
Array	`JsonArray`, `JsonArrayConst`
Object	`JsonObject`, `JsonObjectConst`

No naked `char`

In C++, `char`, `signed char`, and `unsigned char` are different times. `char` is meant to store characters, whereas `signed char` and `unsigned char` are meant to store numbers.

ArduinoJson supports `signed char` and `unsigned char`, but not `char`.

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, you can call `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 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, I'll pretend there are only two repositories, so it takes less space in the book.

Here is the new sample JSON document:

```
[
  {
    "name": "ArduinoJson",
    "stargazers": {
      "totalCount": 6287
    },
    "issues": {
      "totalCount": 22
    }
  },
  {
    "name": "pdfium-binaries",
    "stargazers": {
      "totalCount": 632
    },
    "issues": {
      "totalCount": 14
    }
  }
]
```

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. Create the `JsonDocument`.
3. Call `deserializeJson()`.

Let's do it:

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

// Create the JsonDocument
JsonDocument doc;

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

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

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

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 access 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, which 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 we've seen the same 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 the 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()`:

```
if (repo666.isNull()) ...
```

Alternatively, you can compare to `nullptr` (but not `NULL`!), like so:

```
if (repo666 == nullptr) ...
```

Also, null `JsonObject` evaluates to `false`, so you can check that it's not null like so:

```
if (repo666) ...
```

A null `JsonObject` looks like an empty object, except 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. 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. In short, this pattern saves you from constantly checking that a result is not null.

If ArduinoJson didn't implement this pattern, we could not write the following statement because any missing value would crash the program.

```
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 in advance.

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? 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 is 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`, they affect the `JsonDocument` too.

3.7.2 Number of elements in an array

The first thing you probably want to know about an array is its number of elements. 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.

3.7.3 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 you must 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 the array contains objects. If it's not your case, you can use a `JsonVariant` instead.

3.7.4 Detecting the type of an element

We test the type of array elements the same way we did for object members: 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 Reading from a stream

In the Arduino jargon, a stream is a volatile data source, like a serial port or a TCP connection. Contrary 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.

The `Stream` abstract class materializes this concept. Here are examples of classes derived from `Stream`:

Library	Class	Well known instances
Core	`HardwareSerial`	`Serial`, `Serial1`…
ESP	`BluetoothSerial`	`SerialBT`
	`File`	
	`WiFiClient`	
	`WiFiClientSecure`	
Ethernet	`EthernetClient`	
	`EthernetUDP`	
GSM	`GSMClient`	
SD	`File`	
SoftwareSerial	`SoftwareSerial`	
WiFi	`WiFiClient`	
Wire	`TwoWire`	`Wire`

`std::istream`

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

ArduinoJson supports both `Stream` and `std::istream`.

3.8.1 Reading from a file

As an example, we'll create a program that reads a JSON file stored on an SD card. We suppose 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<const char*>());
  Serial.println(repo["stargazers"]["totalCount"].as<int>());
  Serial.println(repo["issues"]["totalCount"].as<int>());
}
```

Remarks:

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 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 in SPIFFS or LittleFS, as we'll see in the case studies.

3.8.2 Reading from an HTTP response

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

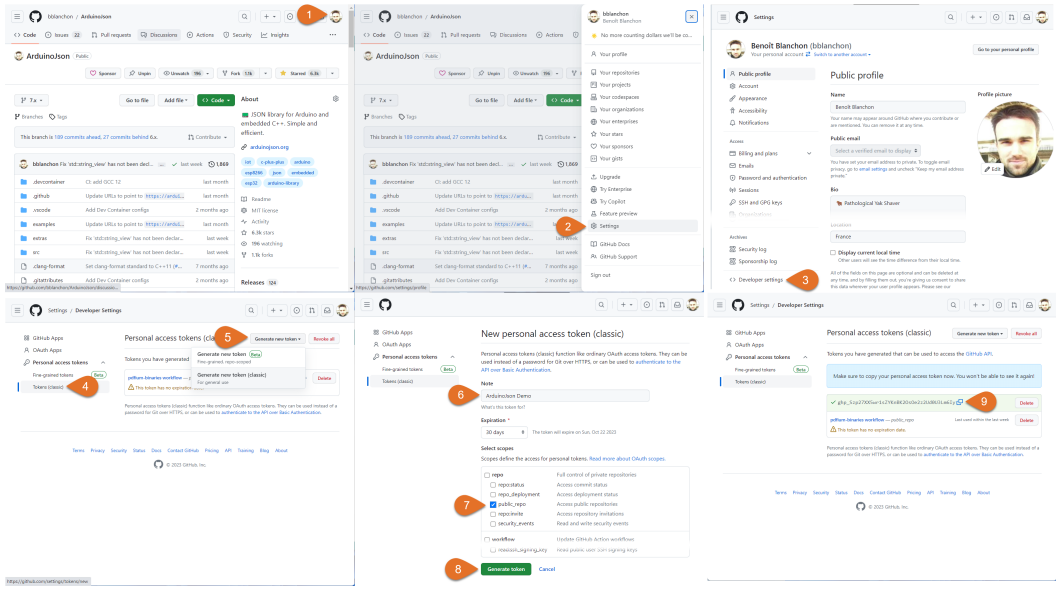
As I said, we need a microcontroller that supports HTTPS, so we'll use an ESP8266 with the library ESP8266HTTPClient. Don't worry if you don't have a compatible board; we'll see other configurations in the case studies.

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. Click on your profile picture.
2. Go to your personal settings.
3. Go in "Developer settings."
4. Go in "Personal access token."
5. Click on "Generate a new token."
6. Enter a name, like "ArduinoJson tutorial."
7. Check the scopes (i.e., the permissions); we only need "public_repo."
8. Click on "Generate token."
9. GitHub shows the token.

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 "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"
```

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 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
```

Certificate validation

Because I don't want to make this example more complicated than necessary, I'll disable the SSL certificate validation like so:

```
WiFiClientSecure client;
client.setInsecure();
```

What could be the consequence? Since the program doesn't verify the certificate, it cannot be sure of the server's authenticity, so it could connect to a rogue server that pretends to be `api.github.com`. This is indeed a serious security breach because the program would send your Personal Access Token to the rogue server. Fortunately, this token has minimal permissions: it only provides access to public information. However, in a different project, the consequences could be disastrous.

If your project presents any security or privacy risk, you must enable SSL certificate validation. `WiFiClientSecure` provides several validation methods. For a simple solution, use `setFingerprint()`, but you'll have to update the fingerprint frequently. For a more robust solution, use `setTrustAnchors()` and make sure your clock is set to the current time and date.

The request

To interact with the new GraphQL API, we must send a `POST` request (instead of 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 a GraphQL query allows you to select the pieces you want within the universe of information 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 can 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(client, "https://api.github.com/graphql");
http.addHeader("Authorization", "bearer " GITHUB_TOKEN));
http.POST("{\"query\":\"{viewer{name,repositories(ownerAffiliations:...");
```

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 (which is the variable `client` in our case). 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 protocol version is part of the request, we must call `useHTTP10()` before calling `POST()`.

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

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

Extracting the values from the JSON document is 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": 6287
            },
            "issues": {
              "totalCount": 22
            }
          },
          {
            "name": "pdfium-binaries",
            "stargazers": {
              "totalCount": 632
            },
            "issues": {
              "totalCount": 14
            }
          },
          ...
        ]
      }
    }
  }
}
```

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:

```
// Prepare the WiFi client
WiFiClientSecure client;
client.setInsecure();

// Send the request
HTTPClient http;
http.begin(client, "https://api.github.com/graphql");
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();

// Deserialize the JSON document in the response
JsonDocument doc;
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(repo["name"].as<const 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 this:

```
ArduinoJson, stars: 6287, issues: 22
pdfium-binaries, stars: 632, issues: 14
ArduinoStreamUtils, stars: 229, issues: 6
ArduinoTrace, stars: 166, issues: 2
WpfBindingErrors, stars: 78, issues: 4
django-htmx-modal-form, stars: 61, issues: 2
dllhelper, stars: 46, issues: 1
cpp4arduino, stars: 41, issues: 1
ArduinoContinuousStepper, stars: 20, issues: 0
django-htmx-messages-framework, stars: 19, issues: 1
```

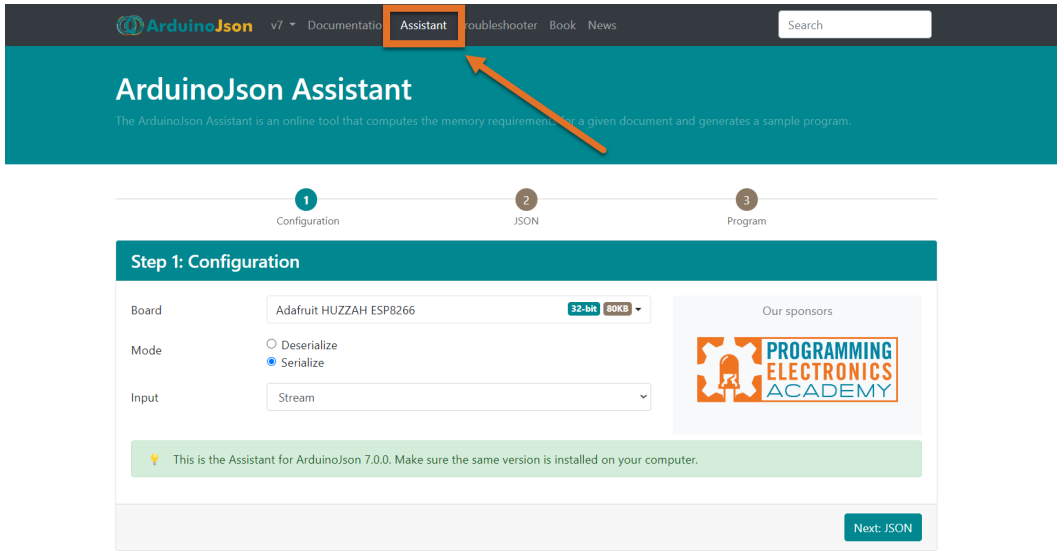
You can find the complete source code of this example in the `GitHub` folder in the zip file provided with the book. Compared to what is shown above, the source code handles the connection to the WiFi network, checks errors, and uses Flash strings when possible.

3.9 The ArduinoJson Assistant

We just learned how to deserialize a JSON document with ArduinoJson, and we saw that extracting the values from a complex document can be tedious.

To simplify this task, I created the **ArduinoJson Assistant**, an online tool that generates the code to deserialize a JSON document.

You can find it at `arduinojson.org/v7/assistant`, but you can also access it from `arduinojson.org` by clicking on the "Assistant" link in the menu.

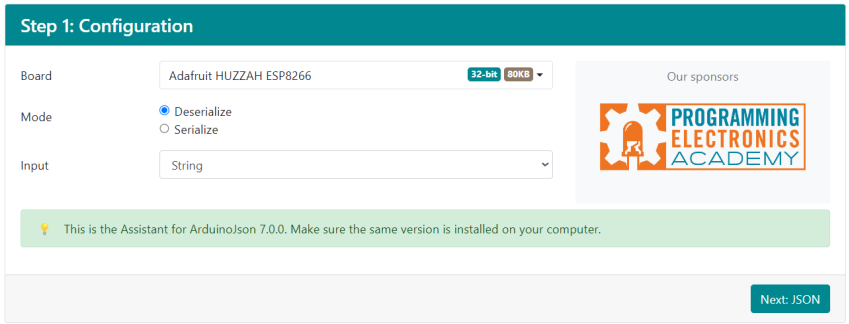


In addition, the ArduinoJson Assistant computes the memory consumption of your program and shows the usage relative to the memory available on your board.

The Assistant for ArduinoJson 7 is composed of 3 steps:

1. Configuration.
2. JSON
3. Program

3.9.1 Step 1: Configuration



In the first step, you must specify the following information:

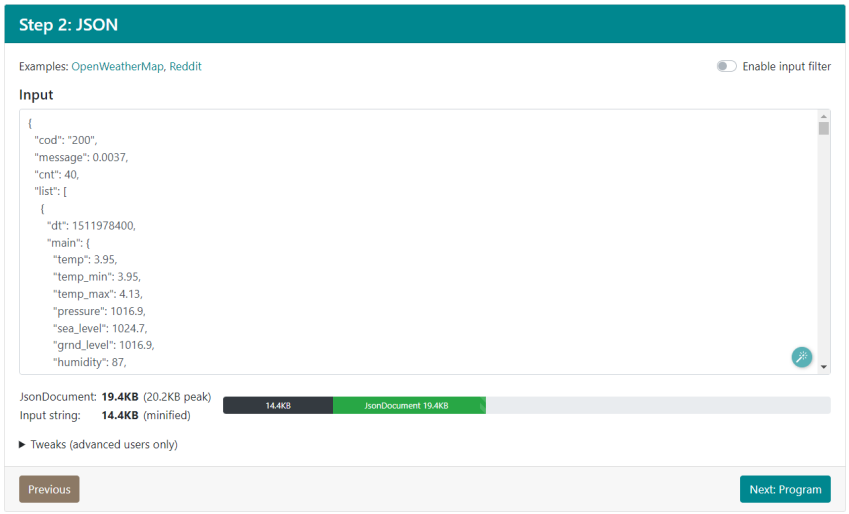
1. The board you are using.
2. Whether you want to serialize or deserialize.
3. The type of input you want to use.

Specifying the board allows the Assistant to know if the microcontroller is 8-bit or 32-bit and the amount of memory available. This information is used to compute the memory consumption of your program and check it is within the limits of your board.

Choosing serialization or deserialization affects the memory consumption, the advanced settings available in the next step, and the code generated in the last step.

The input type can be either a `char` pointer, a string class, or a stream. This setting affects the memory consumption and the code generated in the last step.

3.9.2 Step 2: JSON



In the second step, you must enter the JSON document you want to deserialize. If you're just trying out the Assistant, you can use one of the examples at the top of the page.

The Assistant will check the validity of the JSON document and display an error message if it's not valid.

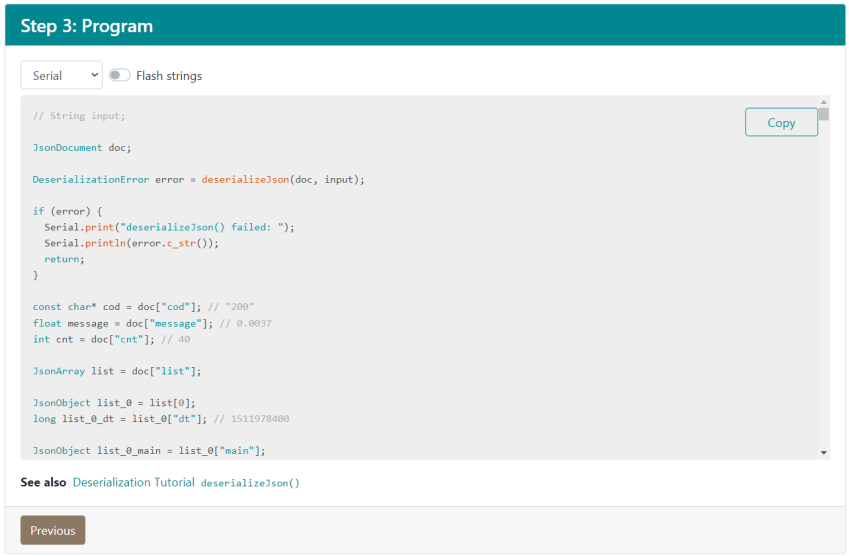
In the upper right corner, you can see the "Enable input filter" checkbox. We'll talk about it in the "Advanced Techniques" chapter.

Below the JSON document, you can see the memory consumption of your program. If the memory consumption is too high, the Assistant will display a warning message.

The Assistant might show other warnings in this step. For example, if your document is deeply nested or contains numberstoo large to fit in a `long`.

At the bottom of this step, you can see the "Tweaks", which are advanced settings that affect the computation of the memory consumption. We'll talk about them in the "Advanced Techniques" chapter.

3.9.3 Step 3: Program



In the last step, the Assistant generates the code to deserialize the JSON document you entered in the previous step.

At the top of the page, you can customize the program, for example, by choosing `std::cout` instead of `Serial` for the output. You can also opt for Flash string for the error messages.

3.10 Summary

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

- `JsonDocument` stores the memory representation of the document.
- `deserializeJson()`:
  - `deserializeJson()` parses the input and fills the `JsonDocument`.
  - `deserializeJson()` returns a `DeserializationError` that you can test with `if (err)`.
  - `deserializeJson()` can read from a `const char*`, `String`, a `std::string`, `Stream`, or `std::istream`.
- `JsonArray` and `JsonObject`:
  - You can extract values 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 but points to it.
  - `JsonVariant` supports implicit conversion, but you can also call `as<T>()`.
- The ArduinoJson Assistant is an online tool that:
  - Computes the memory consumption of your program.
  - Checks that the memory consumption is within the limits of your board.
  - Generates the code to deserialize a JSON document.

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 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 example for this chapter: pushing data to Adafruit IO.

Adafruit IO is a cloud storage service for IoT data: it collects the data coming from your devices and can trigger some action, such as sending an email when a criteria is met.

They offer a free plan with the following restrictions:

- 30 data points per minute
- 30 days of data storage
- 10 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 the integer we want to save in Adafruit IO.
2. `"lat"` is the latitude coordinate.
3. `"lon"` is the longitude coordinate.

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

4.2.2 Creating the JsonDocument

As for the deserialization, we start by creating a `JsonDocument` to hold the memory representation of the object. The previous chapter introduced `JsonDocument`, so I assume you're familiar with it.

```
JsonDocument doc;
```

The `JsonDocument` is currently empty, and `JsonDocument::isNull()` returns `true`. If we serialized it now, the output would be "`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;
```

If there is not enough memory to store a new value in the `JsonDocument`, the new value is silently ignored. However, you can detect if some values are missing by checking `JsonDocument::overflowed()`, which returns `true` when an allocation failed.

To be honest, I never check `JsonDocument::overflowed()` 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.

4.2.4 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 members to add? What if you want to create an empty object?

When you need an empty object, you can no longer rely on the implicit conversion. Instead, you must explicitly convert the `JsonDocument` to a `JsonObject` 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 serialized this document, the output would be "`{}`".

4.2.5 Replacing and removing members

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

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

Finally, you can remove a member with `JsonObject::remove(key)`, for example:

```
// Remove the "value" field
obj.remove("value ");
```

4.3 Creating an array

4.3.1 The example

Now that we can create objects, let's see how to 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 Adding elements

In the previous section, we saw that an empty `JsonDocument` automatically becomes an object as soon as we insert the first member. This statement was only partially correct: 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, this happens if we call `JsonDocument::add()` like so:

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

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

Alternatively, we can create the same array with the `[]` operator like so:

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

However, this second syntax is a little slower because it requires walking the list of members. Use this syntax to *replace* elements and `add()` to append elements to the array.

Now that we can create an array, let's rewind a little because that's not the JSON array we want: instead of two integers, we need two nested objects.

4.3.3 Adding nested objects

To add the nested objects to the array, we call `JsonArray::add<JsonObject>()`. This function returns a reference to the newly created object, so we can use the subscript operator (`[]`) to add members.

Here is how to create our sample document:

```
JsonObject obj1 = doc.add<JsonObject>();
obj1["key"] = "a1";
obj1["value"] = analogRead(A1);

JsonObject obj2 = doc.add<JsonObject>();
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);
```

Again, this syntax is slower because it needs to walk the list, so only use it for small documents.

4.3.4 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` serializes to `[]`.

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

4.3.5 Replacing and removing elements

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

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

Finally, you can remove an element from the array with `JsonArray::remove()`:

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

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'm not a big fan of this class, so instead, we'll use a plain old C string:

```
// 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);
```

After this call, 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 buffer size

If you're a C programmer, you may have been surprised I didn't provide the buffer size 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 buffer size 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, so 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 truncated.

4.4.4 Measuring the length

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

1. allocating an output buffer,
2. reserving the size on disk,
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 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, so prefer minified documents.

4.4.5 Writing to a String

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

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

Of course, this also works with `std::string`.

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. This feature also works for `String`, except the behavior is slightly different.

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.

For example, we could rewrite the previous snippet like this:

```
// Cast the JsonDocument to a string
String output = 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.

In the previous chapter, we saw 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`.

Performance issues

`serializeJson()` writes bytes one by one to the output stream, which can result in bad performances with unbuffered streams like `WiFiClient` or `File`.

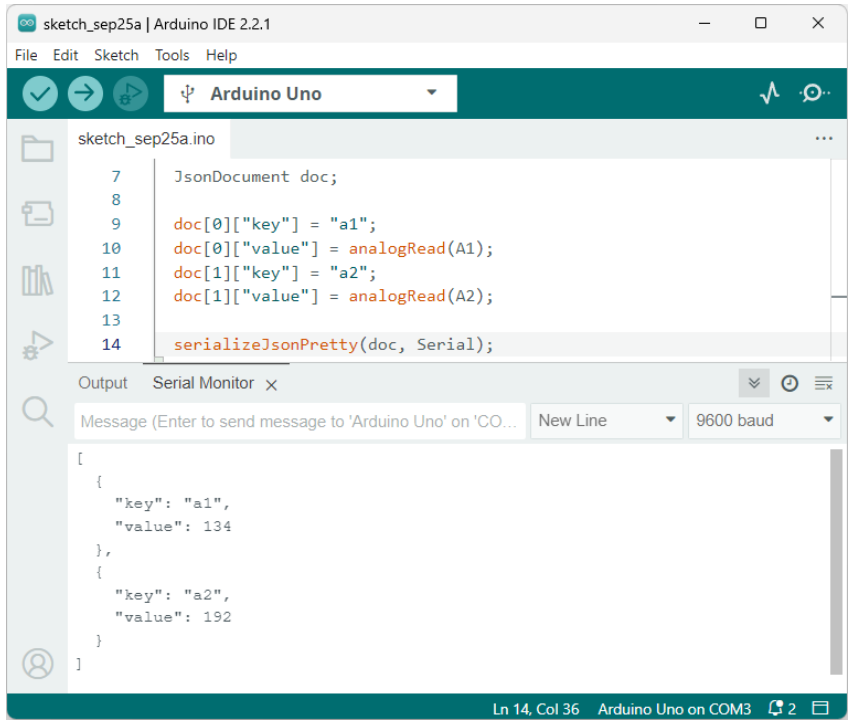
We'll see a simple workaround in the next chapter.

4.5.2 Writing to the serial port

The most famous implementation of `Print` is `HardwareSerial`, 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.



If you want to send JSON documents between two boards, I recommend using `Serial1` for the communication link and keeping `Serial` for the debugging link. Of course, this requires that your board has several UARTs, which is not the case of our UNO R3.

Alternatively, you can use `Wire` for the communication link, but you must know that the Wire library limits the size of a message to 32 bytes (but there is a workaround for longer messages).

In theory, `SoftwareSerial` could also serve as the communication link, but I highly recommend against it because it's completely unreliable.

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 SPIFFS or LittleFS, as we'll see in the case studies.

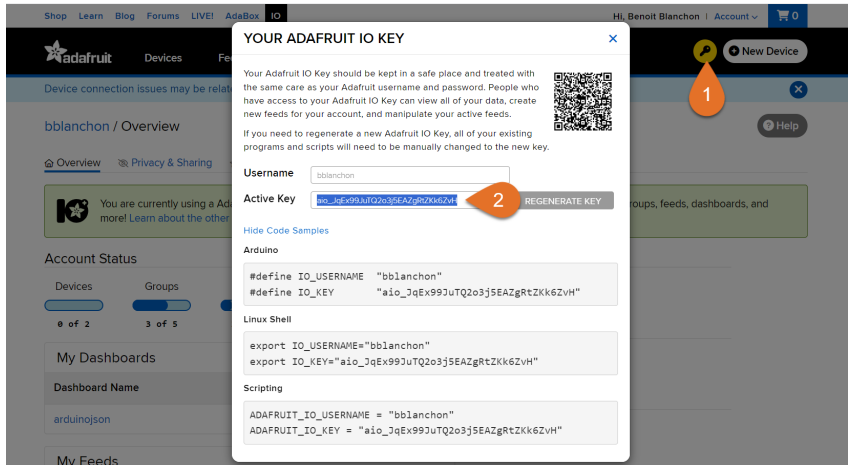
4.5.4 Writing to a TCP connection

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

As I said in the introduction, we'll suppose our program runs on an Arduino UNO with an Ethernet shield.

Preparing the Adafruit IO account

To run this program, you need an account on Adafruit IO (a free account is sufficient).



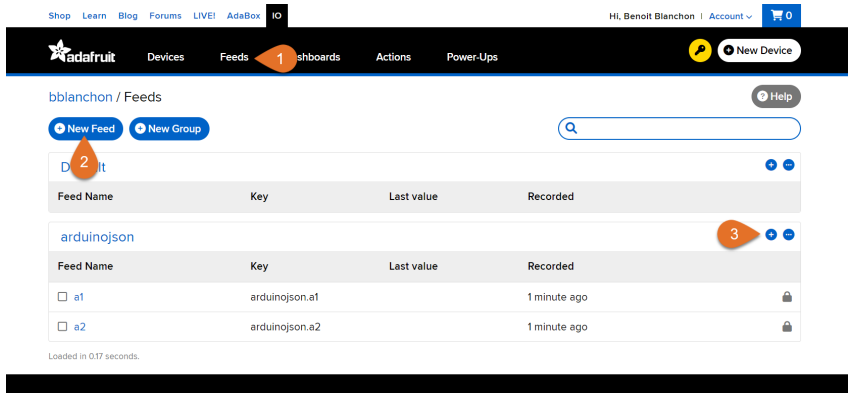
Then, you must copy your user name and "AIO key" to the source code.

```
#define IO_USERNAME "bblanchon"
#define IO_KEY "aio_JqEx99JuTQ2o3j5EAZgRtZKk6ZvH"
```

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

```
X-AIO-Key: aio_JqEx99JuTQ2o3j5EAZgRtZKk6ZvH
```

Finally, you need to create a "group" named "arduinojson" in your Adafruit IO account. In this group, you must 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 example 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/1.0
Host: io.adafruit.com
Connection: close
Content-Length: 103
Content-Type: application/json
X-AIO-Key: aio_JqEx99JuTQ2o3j5EAZgRtZKk6ZvH

{"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 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.

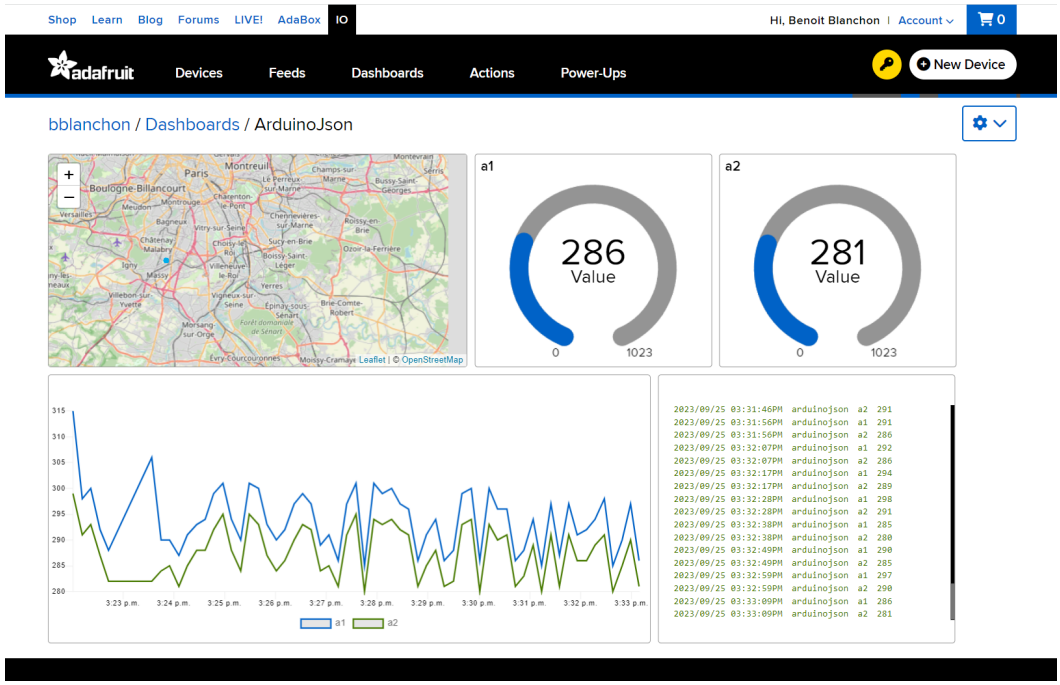
Here is the code:

```
// Create an empty document
JsonDocument doc;
```

```
// Add the "location" object
JsonObject location = doc["location"].to<JsonObject>();
location["lat"] = 48.748010;
location["lon"] = 2.293491;

// Add the "feeds" array
JsonArray feeds = doc["feeds"].to<JsonArray>();
JsonObject feed1 = feeds.add<JsonObject>();
feed1["key"] = "a1";
feed1["value"] = analogRead(A1);
JsonObject feed2 = feeds.add<JsonObject>();
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/1.0"
client.println("POST /api/v2/" IO_USERNAME
               "/groups/arduinojson/data HTTP/1.0");

// 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 code I removed from the book 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 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
`const char*`	pointer
`char*`	copy
`String`	copy
`const __FlashStringHelper*`	copy

ArduinoJson will store only one copy of each string, a feature called "string deduplication". For example, if you insert the string `"hello"` multiple times, the `JsonDocument` will only keep one copy.

4.6.1 An example

Compare this program:

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

with the following:

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

They both produce the same JSON document, but the second one consumes more memory because ArduinoJson copies the strings. For example, on an 8-bit microcontroller, the array consumes an additional 12 bytes. On a 32-bit microcontroller, it would take 30 extra bytes.

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 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")];
```

4.7 Inserting special values

Before finishing this chapter, let's see how we can insert special values in the JSON document.

4.7.1 Adding null

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
arr.add(nullptr);

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

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

4.7.2 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]
]
```

Use this feature when a part of the document cannot change; it will simplify your code and reduce the executable size. You can also use it to insert something the library doesn't allow.

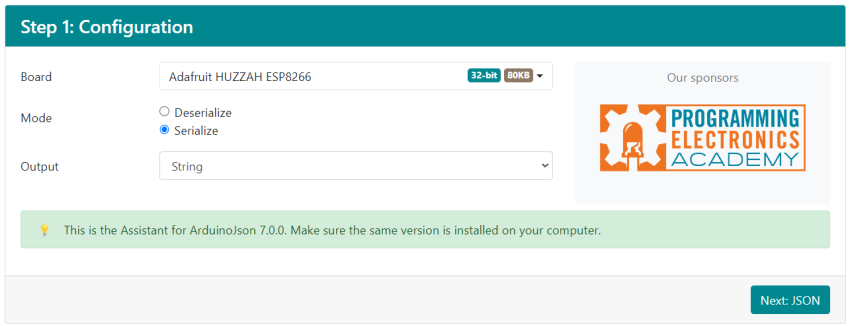
You can pass a Flash string or a `String` instance to `serialized()`. As usual, Flash strings must have the type `const __FlashStringHelper*` to be recognized as such.

Unlike regular ones, strings marked with `serialized()` are always stored by copy, even if they are `const char*`.

4.8 The ArduinoJson Assistant

In the previous chapter, we saw how the ArduinoJson Assistant could help us deserialize a JSON document. Now, we'll see how it can help us serialize a JSON document.

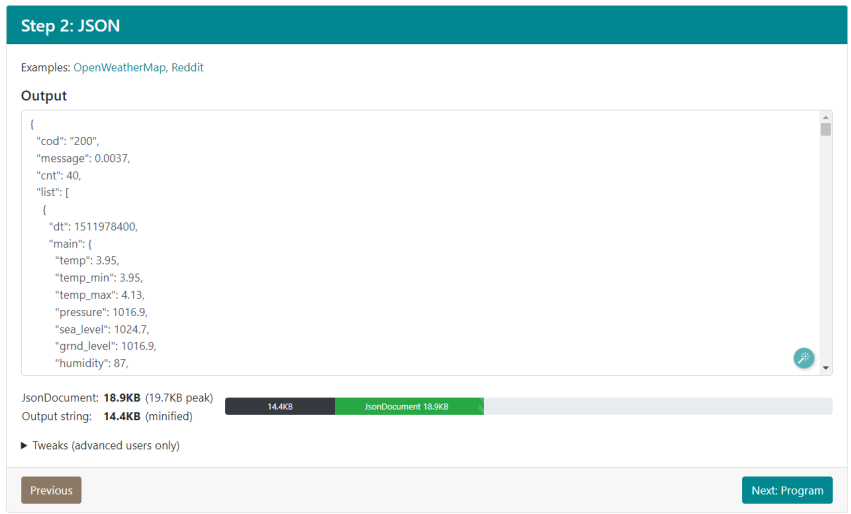
4.8.1 Step 1: Configuration



As we saw in the last chapter, the first step is to choose the board and the mode (serialization or deserialization), but instead of choosing the input type, we must choose the output type.

Again, these settings affect the memory consumption and the code generated in the last step.

4.8.2 Step 2: JSON



Step 2 changes slightly between serialization and deserialization. First, there is no filter option for serialization, and second, the advanced settings hidden in the "Tweaks" section are different. Again, we'll talk about them in the next chapter.

4.8.3 Step 3: Program

In the last step, the Assistant generates the code to serialize the JSON document you entered in the previous step.

You'll notice that contrary to deserialization, the Assistant doesn't offer any customization for the output. However, it writes the program according to the output type you chose in the first step.

4.9 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 (`obj[key] = value`).
    - Call `obj[key].to<JsonArray>()` to add a nested array.
    - Call `obj[key].to<JsonObject>()` to add a nested object.
  - The first time you add a member to a `JsonDocument`, it automatically becomes an object.
  - To append an element to an array, call `add()`.
    - Call `arr.add<JsonArray>()` to append a nested array.
    - Call `arr.add<JsonObject>()` to append a nested 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`, which invalidates all previously acquired references.
  - `JsonDocument::to<T>()` returns a reference to the root array or object.
  - 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`, `WiFiClient`, or `File`) to `serializeJson()` to avoid a copy in the RAM.

- The ArduinoJson Assistant is an online tool that:
  - Computes the memory consumption of your program.
  - Checks that the memory consumption is within the limits of your board.
  - Generates the code to serialize a JSON document.

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 you could use in your projects. These techniques cover serialization, deserialization, or both. Most apply to every project; others are only usable with specific hardware.

Unlike the previous chapter, there won't be a complete example here. Instead, I'll only show small snippets to demonstrate how to use each technique. In the "Case Studies" chapter, however, we'll see complete programs that leverage some of these techniques.

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 contained just what we wanted and nothing more.

In most cases, however, web services return a response containing way too much information. For example, OpenWeatherMap, which we'll explore in the case studies, returns JSON documents containing 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

Unfortunately, reading a large document containing many uninteresting values is 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 stencil to filter the input. This document must contain the value `true` for each member you want to keep. For arrays, create only one element; it will serve as a template for all the elements.

Once the filter is ready, wrap it with `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.

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": 6287
            },
            "issues": {
              "totalCount": 22
            }
          },
          {
            "name": "pdfium-binaries",
            "stargazers": {
              "totalCount": 632
            },
            "issues": {
              "totalCount": 14
            }
          },
          ...
        ]
      }
    }
  }
}
```

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

```
{
  "data": {
    "viewer": {
      "repositories": {
        "nodes": [
```

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

Compare this filter with the input document, and you'll see I replaced the first instance of `name` with the value `true`, and I removed everything else. Indeed, when the filter document contains an array, `deserializeJson()` uses the first element to filter all the elements from the input array, ignoring the others.

Here is how this filter translates into code:

```
// Create the filter document
JsonDocument filter;
filter["data"]["viewer"]["repositories"]["nodes"][0]["name"] = true;

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

After constructing the filter, we wrap it with `DeserializationOption::Filter` before passing it to `deserializeJson()`. There is nothing else to change; the rest of the code remains the same.

Now, if you call `serializeJsonPretty()` to see what the document looks like, you will see:

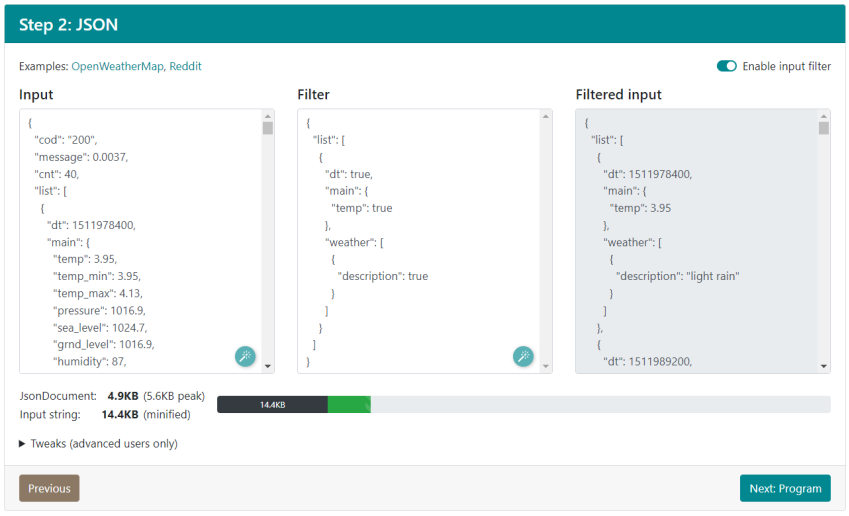
```
{
  "data": {
    "viewer": {
      "repositories": {
        "nodes": [
          {
            "name": "ArduinoJson",
          },
          {
```

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

As you can see, only the `"name"` member was kept, and all other values were instantly discarded.

ArduinoJson Assistant

The ArduinoJson Assistant can help you design the filter document. Select the "Deserialize" mode in the first step and turn on the "Filter" option in the second step. The interface will split into three columns: the input, the filter, and the filtered input.



You can edit the filter document in the middle column, and the Assistant will update the filtered input in the right column.

Notice that the memory consumed by the `JsonDocument` is significantly reduced: it goes from 21.3KB to 5.6KB in the OpenWeatherMap example. The "peak" memory consumption (6.2KB in the screenshot) takes the excluded strings into account because the parser stores them temporarily before discarding them.

Limitations

The filtering feature has the following limitations:

1. It cannot exclude array elements, only members of objects.
2. It cannot exclude members conditionally (e.g., "keep `person` object only if `age` is greater than `18`").

Some query languages, like JSONPath, would allow you to do that, but ArduinoJson does not support them, and there is currently no plan to add such a feature.

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 of 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": 6287
            },
            "issues": {
              "totalCount": 22
            }
          },
          {
            "name": "pdfium-binaries",
            "stargazers": {
              "totalCount": 632
            },
            "issues": {
              "totalCount": 14
            }
          },
          ...
        ]
      }
    }
  }
}
```

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" array, 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 this 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.

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

// Print the content
Serial.print(" - ");
Serial.print(doc["name"].as<const 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 replace it with the next element.

Jumping to the next element

When `deserializeJson()` returns, the next character in the stream should be a 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 the "deserialization in chunks" technique:

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

// Repeat for each element of the array
do {
  // Deserialize one element of the node array
  JsonDocument doc;
  deserializeJson(doc, response);

  // Print the content
  Serial.print(" - ");
  Serial.print(doc["name"].as<const 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 have `char*` parameters, but they should be `const char*`. 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, but 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 have 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 to send and receive a stream of JSON objects.

Principle

In the previous section, we used 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 leverage this feature to deserialize a continuous stream of JSON objects. For example, imagine we transmit instructions to our device via the serial port. Each instruction is a JSON object that contains the details 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 how 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 and times out if nothing comes. To avoid this timeout, we must wait until some data is available before calling `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, we can call `deserializeJson()` and perform the requested action.

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

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

  // Read the next instruction
  JsonDocument doc;
  deserializeJson(doc, Serial);

  // Perform the 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 we saw, reading a JSON stream is fairly straightforward. Well. Writing is even simpler. All we 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 Using external RAM

Motivation

Microcontrollers have a small amount of RAM, but sometimes, you can add an external chip to increase the total capacity. For example, many ESP32 boards embed an external PSRAM connected to the SPI bus. This chip adds up to 4MB to the original 520kB of the ESP32.

Depending on the configuration, the program may use the external RAM implicitly or explicitly.

- With the *implicit* mode, the standard `malloc()` uses the internal and the external RAM. This behavior is entirely transparent to the application.
- With the *explicit* mode, the standard `malloc()` only uses the internal RAM. The program must call dedicated functions to use the external RAM.

Because it's transparent, the *implicit* mode is easier to use. Unfortunately, the external RAM is much slower than the internal RAM, so mixing the two might slow down the whole application.

When performance matters, it's better to use the *explicit* mode, which means we cannot use the standard functions. Therefore, classes like `JsonDocument`, which call the regular `malloc()` and `free()`, cannot use the external RAM (at least not by default).

Principle

`JsonDocument`'s constructor has a parameter of type `ArduinoJson::Allocator*` that allows you to specify a custom allocator. This parameter is optional, and if you don't provide it, the document uses the following allocator:

```
class DefaultAllocator : public Allocator {
 public:
  void* allocate(size_t size) override {
    return malloc(size);
  }
```

```
  void deallocate(void* ptr) override {
    free(ptr);
  }

  void* reallocate(void* ptr, size_t new_size) override {
    return realloc(ptr, new_size);
  }
};
```

As you can see, the allocator class simply forwards the calls to the appropriate functions.

To use the external RAM instead of the internal one, we must create a new allocator class that calls the proper functions.

Implementation

I'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 use 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 defines the required features of the 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:

```
class SpiRamAllocator : public Allocator {
 public:
```

```
  void* allocate(size_t size) override {
    return heap_caps_malloc(size, MALLOC_CAP_SPIRAM);
  }

  void deallocate(void* ptr) override {
    heap_caps_free(ptr);
  }

  void* reallocate(void* ptr, size_t new_size) override {
    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 the allocator
SpiRamAllocator allocator;

// Create a JsonDocument using the allocator
JsonDocument doc(&allocator);

// Use the document as usual
deserializeJson(doc, input);
```

As you can see, we pass the allocator by address, hence the `&` operator. You can (and should) use the same allocator for several documents.

5.6 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 appreciate this kind of code because it minimizes memory consumption, but 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, but 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 to easily debug the program.

Principle

The statement `serializeJson(doc, wifiClient)` calls the function `serializeJson(const JsonDocument&, Print&)`. This function takes an instance of `Print`, an abstract class representing the "output stream" concept 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 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 to 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 an 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 install.

Then, 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 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 used `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.

5.7 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 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 RAM.

Similarly, `deserializeJson()` reads a stream one byte at a time. This feature is crucial for deserializing in chunks and JSON streaming. Unfortunately, it hurts the performance with some implementations of `Stream`.

In this section, we'll learn how to add buffering to ArduinoJson and improve 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` instance 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 in 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` instance with `ReadBufferingStream`. As previously, the constructor of `ReadBufferingStream` takes two `Stream&` parameters: the stream to decorate and the size of the buffer.

```
// Add buffering to "file"
ReadBufferingStream bufferedFile(file, 64);

// Read the JSON document in 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 bother direction: read and write
BufferingStream bufferedClient(wifiClient, 64);
```

```
// Receive the JSON document in chunks of 64 bytes
deserializeJson(doc, bufferedClient);

// Send the JSON document in chunks of 64 bytes
serializeJson(doc, bufferedClient);
bufferedClient.flush();
```

The StreamUtils library offers many other options; please check out the documentation.

5.8 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. This overhead is negligible most of the time, but if performance is crucial, it's better to avoid virtual calls.

In this section, we'll see how to 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, the processor must look up the function's address.

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 function is short, the overhead can become significant. Not only does the lookup cost a few extra cycles, but we also miss the opportunity for 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 can see, no virtual functions are 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);
};
```

Yes, the names, parameters, and return types of these methods are inconsistent, but I wanted to mirror the ones from `Print` and `Stream`.

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, which is the best on this topic.

Custom writer

Suppose we created a file with `fopen()` and want to write a JSON document. 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: 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 to 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.

5.9 Custom converters

Motivation

It's very common to insert a timestamp in a JSON document. The usual way to do so is to call the standard `strftime()` function. As `snprintf()`, this function takes a destination buffer and a format string (although the format specification is different, of course).

For example, here is how we could insert the current date and time in a JSON document:

```
// Get the current time and extract a tm struct
time_t now = time(NULL);
tm timestamp = *gmtime(&now);

// Convert tm struct to a string
char buf[32];
strftime(buf, sizeof(buf), "%Y-%m-%dT%H:%M:%SZ", &timestamp);

// Create JSON document
JsonDocument doc;
doc["timestamp"] = buf;
serializeJson(doc, Serial);
```

On the first line, we get the timestamp as an integer, which we then convert into a `tm` instance. The `tm` structure is defined in `<time.h>`; it contains a field with the year number, the month number, etc. Then, the program calls `strftime()` to convert the `tm` structure into a string. Finally, it creates a `JsonDocument`, inserts the values, and prints something like this:

```
{"timestamp":"2023-11-06T19:55:25Z"}
```

To read the timestamp back, we can call `strptime()`, which parses time strings. This function is not standard but is frequently available on Unix and in some Arduino cores. In works like `sscanf()`: it takes an input string and a format specification.

Here is how we could extract a timestamp from a JSON document:

```
// Deserialize JSON document
JsonDocument doc;
deserializeJson(doc, "{\"timestamp\":\"2023-11-06T19:55:25Z\"}");

// Extract a tm struct from the time string
tm timestamp;
strptime(doc["timestamp"], "%Y-%m-%dT%H:%M:%SZ", &timestamp);
```

This `strftime()`/`strptime()` gymnastic works fine but introduces a lot of boilerplate code. It's alright if we only do this in one place, but it's not suitable to manipulate several timestamps.

In this section, we'll see how we can teach ArduinoJson to convert timestamps to strings and vice-versa.

Principle

ArduinoJson allows us to augment the list of supported types through custom converters. These converters are just functions with fixed names and signatures.

Here are the three functions that you must implement to fully support a given type `T`:

- `void convertToJson(const T& src, JsonVariant dst)`
- `void convertFromJson(JsonVariantConst src, T& dst)`
- `bool canConvertFromJson(JsonVariantConst src, const T&)`

ArduinoJson calls `convertToJson()` when you insert a value of type `T` in a `JsonDocument`. This function must convert the source value and set the result in the `JsonVariant`.

`convertFromJson()` is called when you extract a value of type `T` from a `JsonDocument`. It must somehow parse the value and set the result in the destination parameter. The input can be any JSON value, including an object, as we'll see later.

Lastly, `canConvertFromJson()` is called by `is<T>()`. It must check the source parameter and return `true` if it can be converted to `T` or `false` otherwise.

These three functions are optional: you only have to implement the ones used in your program. For example, if you're only reading JSON documents, `convertFromJson()` is sufficient; you don't need to implement `convertToJson()`. Similarly, if your program never calls `is<T>()`, you can omit `canConvertFromJson()`.

However, these functions require that `T` is default-constructible, i.e., that `T` has a default constructor (a constructor with no parameter). Most types are default-constructible, but we'll see how to deal with such cases at the end of this section.

Implementation

Default-constructible types

Let's see how we can apply custom converters to the date and time problem. The `tm` structure is default constructible, so we can use the conversion functions. We'll start with `convertToJson()`, which converts a `tm` structure to a string. Here is the definition:

```
void convertToJson(const tm& src, JsonVariant dst) {
  char buf[32];
  strftime(buf, sizeof(buf), "%Y-%m-%dT%H:%M:%SZ", &src);
  dst.set(buf);
}
```

As you can see, this function calls `strftime()` to convert the `tm` structure into a string and then sets the `JsonVariant` with this string.

Once this function is declared, we insert a `tm` structure into a `JsonDocument`, like so:

```
doc["timestamp"] = timestamp;
```

How does ArduinoJson find the conversion functions?

When a C++ program calls a function, the compiler first looks for a candidate in the current namespace. If it doesn't find any, it searches in the namespaces of the function *arguments*. This feature is called argument-dependent lookup, or ADL.

In our case, we declared the `convertToJson()` in the global namespace, which turns out to be the namespace of the `tm` structure. We could also declare `convertToJson()` in the namespace `ArduinoJson` since it contains `JsonVariant`.

We can write `convertFromJson()`, which performs the reverse operation, like so:

```
void convertFromJson(JsonVariantConst src, tm& dst) {
  const char* timestring = src.as<const char*>();
  if (timestring)
    strptime(timestring, "%Y-%m-%dT%H:%M:%SZ", &dst);
  else
    memset(&dst, 0, sizeof(tm));
}
```

This function extracts the string from the source and calls `strptime()` to fill the `tm` structure. If the string is null (which could happen if the `JsonVariantConst` is null or if it points to a value of a different type), `convertFromJson()` doesn't call `strptime()` but clears the `tm` structure.

We can now extract a `tm` structure from a `JsonDocument`, like so:

```
timestamp = doc["timestamp"];
```

Optionally, we can write `canConvertFromJson()`, which tests if the JSON value can be converted to a `tm` structure. I think checking that the value is a string is sufficient for this example. Here we go:

```
bool canConvertFromJson(JsonVariantConst src, const tm&) {
  return src.is<const char*>();
}
```

This function allows us to call `is<tm>()`. ArduinoJson doesn't call `canConvertFromJson()` before `convertFromJson()`, so we must still check that the string isn't null before calling `strptime()`.

You can find the complete time conversion example in the folder `TimeConverter` of the zip file provided with the book. As you'll see, I moved the tree conversion functions to a dedicated `.cpp` file so you can easily reuse them.

Non-default-constructible types

`convertFromJson()`'s second parameter is a reference to `T`, the type we want to convert. Why does it take a reference when we could use the return value? Because ADL doesn't work for return values but only with arguments.

Internally, ArduinoJson creates an instance of `T` before passing it to `convertFromJson()`. It creates this instance via its default constructor, i.e., without passing any argument. Obviously, this instantiation can only work if the default constructor exists. In the rare cases where the default constructor is missing, we cannot rely on `convertFromJson()`.

To support a non-default-constructible type `T`, we must instead specialize the template class `Converter<T>` and implement the three following *class* functions:

- `void toJson(const T& src, JsonVariant dst)`
- `T fromJson(JsonVariantConst src)`
- `bool checkJson(JsonVariantConst src)`

`Converter<T>` must be defined in the `ArduinoJson` namespace.

For example, suppose that we have the following class defined in our application:

```
class Complex {
  double _real, _imag;
 public:
  explicit Complex(double r, double i) : _real(r), _imag(i) {}
  double real() const { return _real; }
  double imag() const { return _imag; }
};
```

As you can see, this class doesn't have a default constructor (the compiler doesn't generate a default constructor because there is a user-defined constructor). To support this type in ArduinoJson, we must create the following specialization of `Converter<T>`:

```
namespace ArduinoJson {
template <>
struct Converter<Complex> {
  static void toJson(const Complex& src, JsonVariant dst) {
    dst["real"] = src.real();
    dst["imag"] = src.imag();
  }

  static Complex fromJson(JsonVariantConst src) {
    return Complex(src["real"], src["imag"]);
  }
```

```
  static bool checkJson(JsonVariantConst src) {
    return src["real"].is<double>() && src["imag"].is<double>();
  }
};
}
```

Don't be afraid by `template<>`; it's just the C++ syntax to declare a specialization of a template class, i.e., to replace its default implementation. Notice, though, that all functions are `static`.

Now, we can insert `Complex` instances in a JSON document like so:

```
JsonDocument doc;
doc["complex"] = Compex(1.2, 3.4);
serializeJson(doc, Serial);
```

The program above produces the following output:

```
{"complex":{"real":1.2,"imag":3.4}}
```

Similarly, we can extract a `Complex` from a JSON document. For a complete example, please see the folder `ComplexConverter` in the zip file.

5.10 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: they 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 containers 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 form, so with a slight performance boost.

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
`deserializeJson()`	`deserializeMsgPack()`
`serializeJson()`	`serializeMsgPack()`
`serializeJsonPretty()`	

Those are the only changes you need for MessagePack.

Implementation

To demonstrate how we can serialize a MessagePack document, I'll adapt one of the examples of the serialization tutorial. Here is 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
JsonDocument 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);
```

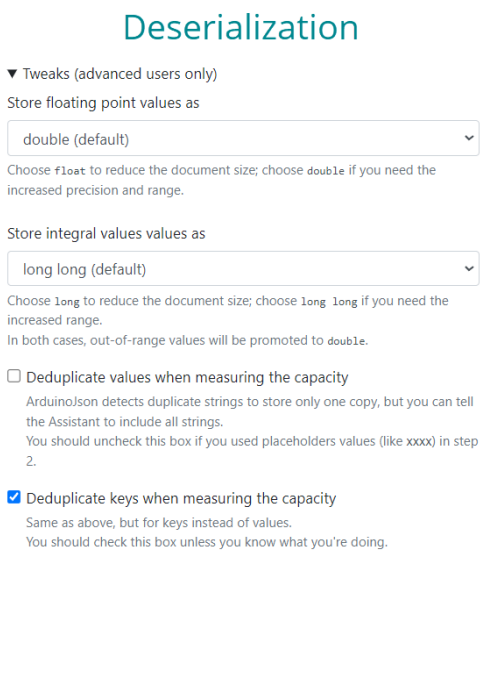
As you see, there is not a lot to say about MessagePack. I personally don't encourage people to use this format. Sure, it reduces the payload size (we saw a reduction of 37% in our example), but the gain is too small to be a game-changer.

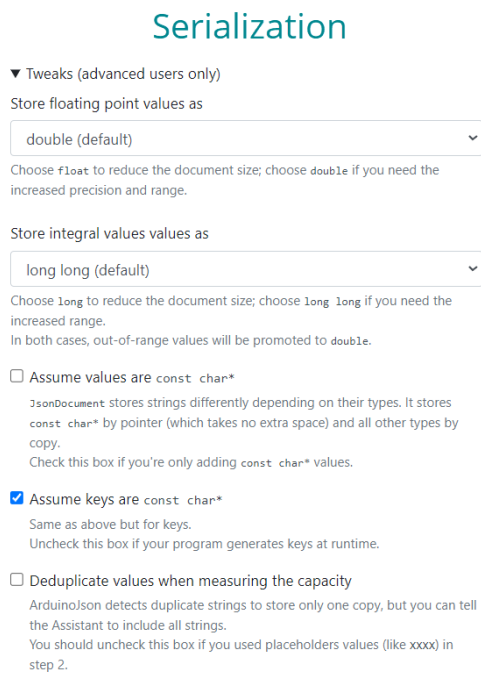
5.11 ArduinoJson Assistant's Tweaks

In the previous chapter, I presented the ArdinoJson Assistant and showed you how you could use this tool to:

1. verify that your microcontroller has enough memory to serialize or deserialize your JSON document
2. write the code to serialize or deserialize your JSON document

You probably notice that the second step of the Assistant contains a collapsible section called "Tweaks". This section contains a few options that control how the Assistant computes the memory consumption. As you can see in the two screenshots below, the available options are different for serialization and deserialization modes.





5.11.1 Floating-point storage

The first option lets you choose the type the `JsonDocument` will use to store floating-point numbers. You can choose between `float` and `double`, and the default is `double`.

If you don't care about the additional precision provided by `double`, you can select `float`, but the memory usage will remain the same unless you also change the next option.

Of course, if you change this value, you must also set the corresponding library flag in your sketch:

```
// Use float instead of double
#define ARDUINOJSON_USE_DOUBLE 0
```

The ArduinoJson Assistant reminds you to do so by displaying a warning in step 3.

5.11.2 Integer storage

The second option lets you choose the type that the `JsonDocument` will use to store integers. You can choose between `long` and `long long`. The default is `long` on 8-bit platforms and `long long` on 32-bit platforms.

As for floating-point numbers, if you don't care about the additional range provided by `long long`, you can select `long`, but the memory usage will remain the same unless you also change the float storage type.

For instance, on a 32-bit microcontroller, the OpenWeatherMap example consumes 20.8KB of RAM with the default settings, and 16.5KB with `float` and `long`. That's a 20% reduction.

Again, if you change this value, you must also set the corresponding library flag in your sketch:

```
// Use long instead of long long
#define ARDUINOJSON_USE_LONG_LONG 0
```

5.11.3 String deduplication

As you probably already know, a `JsonDocument` stores only one copy of each string. We call this feature "string deduplication," and it cannot be disabled.

The Arduino Assistant takes string deduplication into account when it computes memory usage. However, it also allows you to disable string deduplication in case you want to know the worst-case memory usage. Two checkboxes control the string deduplication in the Assistant: the first affects only string values, and the second affects only object keys. Both are enabled by default.

For example, you might disable values deduplication if your JSON document contains placeholders instead of actual data. Let me illustrate this with an example:

```
{
  "networks": [
    {
      "ssid": "XXXXXXXXXXXXXXXX",
      "pass": "XXXXXXXXXXXXXXXX"
    },
    {
      "ssid": "XXXXXXXXXXXXXXXX",
      "pass": "XXXXXXXXXXXXXXXX"
    },
    {
      "ssid": "XXXXXXXXXXXXXXXX",
      "pass": "XXXXXXXXXXXXXXXX"
    }
  ]
}
```

As you can see, this document contains placeholders instead of actual SSIDs and passphrases. All the placeholders are identical, so the Assistant will deduplicate them and compute a size of 268 bytes. However, this memory usage is unrealistic because the input document will never have duplicate SSIDs and passphrases.

In cases like this, you should disable the value deduplication to get a more realistic memory usage. In the example above, the Assistant will compute a size of 393 bytes, which is closer to what the program will actually use.

5.11.4 `const char*` strings

The last couple of tweaks are only available on serialization and affect the way the Assistant computes the memory usage of strings.

You remember from the tutorial that when you insert a string in a `JsonDocument`, it copies the string, except if it's a `const char*`. In this case, the library stores a pointer to the original string. This avoids copying string literals.

For example, the following program stores `"hello"` by address and `"world"` by copy:

```
JsonDocument doc;
doc["hello"] = String("world");
```

The ArduinoJson Assistant cannot guess how your program will insert the strings, so it makes the following assumptions: object keys are `const char*`, and other strings are copied (as in the example above). The rationale is that you are more likely to use string literals as object keys because they are usually constant, but you probably generate the other strings at runtime.

If these assumptions are incorrect, you can change the behavior of the Assistant by checking the corresponding checkboxes. For instance, if you know that your program will generate keys at runtime, you should uncheck "Assume keys are `const char*`". If we do that for the OpenWeatherMap example, the size of the `JsonDocument` rises from 18.9KB to 19.4KB, which is a 2.6% increase.

Notice that if you uncheck one of these options, the Assistant will show the corresponding "Deduplicate keys/values when measuring the size" checkbox.

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 timeout.
- Using external RAM:
  - Define an allocator class that derives from `ArduinoJson::Allocator`.
  - Pass a pointer to the allocator to `JsonDocument`'s constructor.
- 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: one for a single character and the other for multiple characters.
  - The reader class requires `read()` and `readBytes()`.
  - No virtual call is involved.
- Custom converters:
  - Default-constructible types can be added by defining `convertToJson()`, `convertFromJson()`, and `canConvertFromJson()`.
  - Non-default-constructible types require a specialization of `ArduinoJson::Converter<T>`.
  - All conversion functions are optional.
- 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 Introduction

In this chapter, we'll look at the internals of ArduinoJson. My goal is to help you understand how the library works so that you can use it more efficiently. I also want this chapter to be a reference for those who wish to read the source code and contribute to the project.

The information contained in this chapter is tied to a specific version of the library and might not apply to future versions. I tried to find the right balance between simplicity and completeness, so I'll skip some details that are not essential to the understanding of the library. For example, I removed most of the gory details from the code snippets; please refer to the source code for the complete picture.

We'll start by looking at how ArduinoJson represents values in memory: first with variants, then array and objects. Then, we'll look at how the library manages memory resources. Finally, we'll see how the library serializes and deserializes documents.

This chapter is not intended to be read by everyone; feel free to skip it.

6.2 Variants

As you know, values in JSON can have different types (string, integer, boolean, etc), and the type of a value can change over time. In the following snippet, the first element for the array is initially a string, then becomes an integer:

```
JsonDocument doc;
doc[0] = "42";
serializeJson(doc, Serial);  // prints ["42"]
doc[0] = 42;
serializeJson(doc, Serial);  // prints [42]
```

Because we must support multiple types, we cannot use regular `const char*` or `int` to store the value. Instead, we use a union of all the possible types:

```
union VariantContent {
  double asFloat;
  bool asBoolean;
  long asSignedInteger;
  unsigned long asUnignedInteger;
  ArrayData asArray;
  ObjectData asObject;
  const char* asLinkedString;
  struct StringNode* asOwnedString;
};
```

Notice that we have two types of integer (signed and unsigned) and two types of string (linked and owned). We'll see later why we need this.

In case you don't know what a `union` is, it's a type that can store different types of values, but only one at a time. You could see it as a `struct` in which all the fields overlap in memory. The size of the `union` is the size of its largest field.

Because a `union` doesn't remember which type it holds, we also need to store this information. We do this with the following enum:

```
enum VariantType {
  VALUE_IS_NULL,
  VALUE_IS_LINKED_STRING,
  VALUE_IS_OWNED_STRING,
  VALUE_IS_BOOLEAN,
  VALUE_IS_UNSIGNED_INTEGER,
  VALUE_IS_SIGNED_INTEGER,
  VALUE_IS_FLOAT,
  VALUE_IS_OBJECT,
  VALUE_IS_ARRAY,
};
```

Finally, we combine the type and the value in a struct:

```
struct VariantData {
  VariantContent content;
  VariantType type;
};
```

This kind of combination is called a "tagged union" or "discriminated union," but the term "variant" is more common in C++ (think of `std::variant`, for example), hence the names `VariantData` and `JsonVariant`.

The size of this structure is decisive for the overall memory consumption since every value uses one. With the default settings, `VariantData` takes 8 bytes on 8-bit processors and 12 bytes on 32-bit processors.

6.3 Integers

As you saw, `VariantContent` has two integer fields: one for signed integers and one for unsigned integers. Both are needed to cover the whole range of possible values.

I showed them as `long` and `unsigned long`, but they can also be `long long` and `unsigned long long` if you set the `ARDUINOJSON_USE_LONG_LONG` flag.

While we are at it, let's talk about the integer overflows. ArduinoJson 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"] = 40000;

doc["value"].as<int8_t>();    // 0
doc["value"].as<uint8_t>();   // 0
doc["value"].as<int16_t>();   // 0
doc["value"].as<uint16_t>();  // 40000
doc["value"].as<int32_t>();   // 40000
doc["value"].as<uint32_t>();  // 40000
```

Of course, this feature also applies when you use the syntax with implicit conversion:

```
doc["value"] = 40000;

int8_t   a = doc["value"];  // 0
uint8_t  b = doc["value"];  // 0
int16_t  c = doc["value"];  // 0
uint16_t d = doc["value"];  // 40000
int32_t  e = doc["value"];  // 40000
uint32_t f = doc["value"];  // 40000
```

It also works with `JsonVariant::is<T>()`:

```
doc["value"] = 40000;
```

```
doc["value"].is<int8_t>();    // false
doc["value"].is<uint8_t>();   // false
doc["value"].is<int16_t>();   // false
doc["value"].is<uint16_t>();  // true
doc["value"].is<int32_t>();   // true
doc["value"].is<uint32_t>();  // true
```

6.4 String

6.4.1 String nodes

As you know, ArduinoJson stores `const char*` by address and other strings by copy, which saves RAM by avoiding copies of string literals. To support both cases, `VariantContent` has two string fields: `asLinkedString` and `asOwnedString`. Here, "linked" means "stored by address", and "owned" means "stored by copy."

Let's see an example of a linked string:

```
JsonDocument doc;
doc.set("hello");
```

In this case, the string literal `"hello"` is stored by address: the pointer is saved in `asLinkedString`, and the type is set to `VALUE_IS_LINKED_STRING`. Now, let's see an "owned" string:

```
JsonDocument doc;
doc.set(String("hello"));
```

In this case, ArduinoJson allocates a buffer to hold the characters of the string. In addition to the buffer, it saves the string length and a reference counters. This is done through the `StringNode` struct:

```
struct StringNode {
  struct StringNode* next;
  uint16_t references;
  unint16_t length;
  char data[1];
};
```

As you can, see `StringNode` contains a pointer to the next `StringNode` to form a linked list. Indeed, `JsonDocument` keeps a list of all the owned strings so it can reuse them when possible. This is how the string deduplication works.

Notice that the `data` field is only one character long in the declaration. In practice, the library allocates a larger buffer that matches the size of the string.

6.4.2 String adapters

While we are on the topic of strings, let's see how ArduinoJson supports several types of strings: not only `const char*` and `String`, but also Flash string.

ArduinoJson does that through the `StringAdapter` class. This template class has a specialization for each string type. It comes with the helper function `adaptString()` that returns the appropriate class for a given string.

Throughout the code, the term "adapted string" is used to refer to a string that has been wrapped in a `StringAdapter`. All the internal functions of ArduinoJson take adapted strings as a template parameter, whereas the public API takes regular strings such as `const char*` or `String`. This is why the most public-facing functions call `adaptString()` before delegation to the internal ones.

6.4.3 Variable-length arrays

All the public-facing functions with a string parameter are duplicated: one overload take a reference, and the other takes a pointer.

Let's take `JsonObject::remove()` for example:

```
template <typename TString>
void remove(const TString& key) const;

template <typename TChar>
void remove(TChar* key) const;
```

You might think the first overload is enough because it would support all types of strings: `const char*`, `String`, `std::string`, and `const __FlashStringHelper*`. Unfortunately, variable-length arrays don't match the first overload; they only match with pointer parameters, hence the second overload.

As a reminder, a variable-length array (or VLA) is an array whose size is unknown at compile time. For example, the following buffer is a VLA:

```
void foo(size_t n) {
  char buffer[n];
  // ...
}
```

VLAs are not supported by the C++ standard, but most compilers support them anyway. Arduino users don't really care about the standard and use VLAs all the time. That's why ArduinoJson supports them even if it's a lot of work.

6.5 Arrays and objects

ArduinoJson stores arrays and objects as linked lists using the `VariantSlot` struct:

```
class VariantSlot {
  VariantData value;
  SlotId next;
  const char* key;
};
```

As you can see, `VariantSlot` points to the next element using an integer, not a pointer; we'll see why later. Depending on the platform, `SlotId` is either a `uint8_t`, `uint16_t`, or a `uint32_t`, limiting the total number of slots to 256, 64K, or 4M.

The `key` field is only used for objects and ignored for arrays. This design allows using the same struct for both arrays and objects, which simplifies the code but wastes a few bytes of memory for each array element. This is a part that is very likely to change in the future.

The structure `CollectionData` is used to store the first and last element of the collection:

```
struct CollectionData {
  SlotId head;
  SlotId tail;
};
```

`ArrayData` and `ObjectData` are two derived classes of `CollectionData`. They don't add any class variables, only methods to manipulate the collection.

As you can see, `CollectionData` doesn't store the number of elements in the collection. Indeed, adding a variable to store the size would increase the size of `VariantData`, affecting all types of values, not just arrays and objects.

`CollectionData` supports iteration through the `begin()` and `end()` methods, which return a `CollectionIterator*`. For example, here is the definition of `JsonArrayIterator`:

```
class JsonArrayIterator {
 public:
  // public facing functions, like operator++, operator*, etc.

 private:
  CollectionIterator iterator_;
  ResourceManager* resources_;
};
```

As you can see, `JsonArrayIterator` also needs a pointer to the `ResourceManager` to access the `VariantSlot` objects. Indeed, the `CollectionIterator` stores a `SlotId`, not a pointer to the slot. We'll come back to this later.

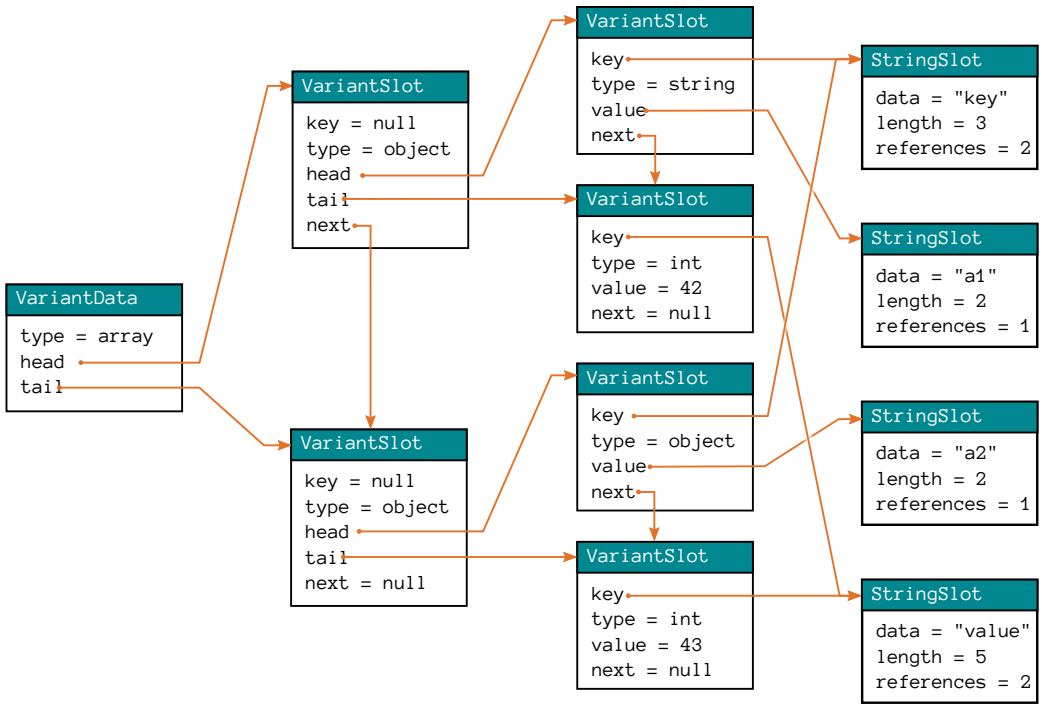
6.6 Document tree

Now that you know how individual pieces are modeled, let's see how a complete JSON document is represented in memory. Let's look at the following example taken from the serialization tutorial:

```
[
  {
    "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, yet its memory representation can be quite complex, as shown in the diagram below.



In this diagram, every box represents an instance of an object, and every arrow represents a pointer.

The left-most box is the root of the tree. It's a `VariantData` with `type` set to `VALUE_IS_ARRAY` and `content.asArray` pointing to the first element and last elements of the array.

These two elements are the two `VariantSlots` in the second column. They both have their `key` set to `nullptr` because they are array elements, and `key` is only used for objects. They both have their `type` set to `VALUE_IS_OBJECT` and `content.asObject` pointing to the first and last members of the object. The `next` field forms a linked list of elements.

The third column contains the four `VariantSlots` representing the two members of each object. This time, we have two types of variants: strings and integers. The string points to the `StringNode` instances of the fourth column. The integers are stored directly in the `VariantSlot`.

Each `StringNode` holds the characters of one of the strings. Notice the reference counter set to `2` for the shared strings.

6.7 Slot pool

As we saw in the previous example, a very simple JSON document already requires a fairly complex tree of objects. In this case, the document required the allocation of six `VariantSlot`s and four `StringNode`s.

`VariantSlot`s are very small objects (8 to 12 bytes, depending on the architecture), so if we were to allocate them individually, we would waste a lot of memory. Indeed, every time we allocate a block of heap memory, the allocator adds a few bytes of overhead to store the size of the block and other information. This overhead is usually 2 to 8 bytes, which is a lot compared to the size of a `VariantSlot`.

To avoid this overhead, ArduinoJson allocates `VariantSlot` in batches of 16 to 128 slots, depending on the architecture. Not only does this avoid the overhead, but it also reduces the number of calls to `malloc()` and `free()`, which is good for performance.

When the pool is exhausted, ArduinoJson allocates a new one, and the cycle continues until the allocation fails or the maximum number of pools is reached.

As we saw earlier, `VariantSlot`, `ArrayData`, and `ObjectData` refer to slots using a `SlotId` instead of a pointer. `SlotId` is an integer composed of the pool index and the index of the slot in the pool. The size of this integer determines the maximum number of pools, as shown in the table below:

Architecture	`SlotId` size	Max slots	Pool size	Max pools
8-bit	1 byte	256	16	16
32-bit	2 bytes	64K	64	256
64-bit	4 bytes	4M	128	32K

Using an integer instead of a pointer has two advantages:

1. It reduces the size of `VariantSlot` since `SlotId` is twice as small as a pointer.
2. It allows reallocating the pools (for example, when `shrinkToFit()` is called) without updating all the pointers.

The classes `VariantPool` and `VariantPoolList` are responsible for managing the pools. The list of pools is implemented as a list of pointers to the pools. This list comes with four preallocated pointers and grows as needed.

When a slot is released, for example, if the value was removed from the `JsonDocument`, the slot is added to a free list. Later, when a new slot is required, we first try to reuse

a slot from the free list; if the free list is empty, we allocate a new slot from the pools. `shrinkToFit()` doesn't release the slots in the free list.

6.8 The resource manager

In the previous chapter, we saw that ArduinoJson never calls `malloc()` or `free()` directly but always delegates to an `Allocator` class. Then, in this chapter, we talked about `VariantPoolList`, which handles the allocation of variants, and `StringPool`, which handles the storage of strings.

These three objects (`Allocator`, `VariantPoolList`, and `StringPool`) are grouped into a single class called `ResourceManager`. This class is responsible for managing the memory of a `JsonDocument`.

```
class ResourceManager {
 public:
  // ...functions to allocate and free string and variants

 private:
  Allocator* allocator_;
  StringPool stringPool_;
  VariantPoolList variantPools_;
};
```

Armed with this knowledge, we can now look at the implementation of `JsonDocument`:

```
class JsonDocument {
 public:
  // ...all the public functions that you love
 private:
  ResourceManager resources_;
  VariantData data_;
};
```

As you can see, `JsonDocument` is simply a wrapper around `ResourceManager` and `VariantData`. The latter is the root of the JSON tree we saw earlier in this chapter.

6.9 Smart pointers

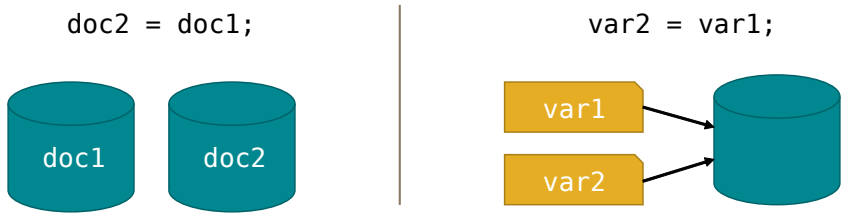
6.9.1 JsonVariant

Now that we know how `JsonDocument` is defined, let's look at `JsonVariant`:

```
class JsonVariant {
 public:
  // ...all the public functions that you love
 private:
  ResourceManager* resources_;
  VariantData* data_;
};
```

As you can see, `JsonVariant`'s implementation resembles `JsonDocument`'s. The only difference is that `JsonVariant` doesn't own the `VariantData` and `ResourceManager` objects; it only stores pointers to them. This makes sense since `JsonVariant` is a smart pointer: it doesn't own the data but only points to it.

If you copy a `JsonDocument`, you get a new document with its own `VariantData` and `ResourceManager` objects. But if you copy a `JsonVariant`, you get a new pointer to the same `VariantData` and `ResourceManager` objects. That's why `JsonVariant` has reference semantics, whereas `JsonDocument` has value semantics.



6.9.2 Unbound JsonVariant

The default constructor of `JsonVariant` sets both pointers to `nullptr`, creating an "unbound" reference. An unbound reference is a reference that doesn't point to any data; it's not the same thing as a `JsonVariant` pointing to a `null` value.

```
JsonDocument doc;
deserializeJson(doc, "{\"hello\":null}");
JsonVariant hello = doc["hello"];  // null reference
JsonVariant world = doc["world"];  // unbound reference
```

The major difference between an unbound and a null reference is that you cannot change the value of an unbound reference, whereas you can change the value of a null reference. As you can see below:

```
hello = "hello";
world = "world";
serializeJson(doc, Serial);  // prints {"hello":"hello"}
```

Indeed, an unbound reference points nowhere, so it has no place to store a value. On the other hand, a null reference points to a `VarianData` set to `null`, so it can store a value.

We'll see how ArduinoJson supports expressions like `doc["world"] = "world"` in the section dedicated to proxies.

There is an undocumented `isUnbound()` function that ArduinoJson uses to differentiate unbound references from null references.

6.9.3 `JsonVariantConst`

In the same vein, we have `JsonVariantConst`, which is a read-only version of `JsonVariant`:

```
class JsonVariantConst {
 public:
  // ...all the public functions that you love
 private:
  const ResourceManager* resources_;
  const VariantData* data_;
};
```

Notice the `const` modifier; that's the only difference with `JsonVariant`. Then, of course, `JsonVariantConst` has fewer public functions than `JsonVariant` since it doesn't allow modification.

6.9.4 `JsonArray` and JsonObject

`JsonArray` and `JsonObject` are very similar to `JsonVariant`, except they point to an `ArrayData` or `ObjectData` instead of a `VariantData`, and the same is true for `JsonArrayConst` and `JsonObjectConst`.

Iteration is implemented by the following classes:

Container class	Iterator class
`JsonArray`	`JsonArrayIterator`
`JsonArrayConst`	`JsonArrayConstIterator`
`JsonObject`	`JsonObjectIterator`
`JsonObjectConst`	`JsonObjectConstIterator`

These iterators are wrappers on top of `CollectionIterator`.

`JsonArray` and `JsonObject` can also be unbound, but there is currently no way to tell the difference between an unbound or a null `JsonArray` or `JsonObject` (no `isUnnound()` function).

6.10 Comparison operators

`JsonVariant` support the comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) for all the built-in types. It also supports comparisons between two `JsonVariant`, which is quite complex since the two variants can have different types.

These comparisons are implemented by the `Comparer` class using the Visitor pattern. `Comparer` is the visitor and `VariantData` is the visited object. When `VariantData` accepts a `VariantComparer`, it calls the appropriate `visit()` function:

- `visit(unsigned long)` if the variant contains an unsigned integer
- `visit(long)` if the variant contains an unsigned integer
- `visit(double)` if the variant contains an unsigned integer
- `visit(bool)` if the variant contains an unsigned integer
- etc.

The comparison between two variants works by calling `accept()` twice: once for each variant. The first call takes a special comparer named `VariantComparer`. Then `VariantComparer` calls `accept()` again, but this time with the regular comparer.

The comparison operators themselves are defined in `VariantOperator`, which also contains the definition of the "or" operator (`|`).

6.11 Converters

ArduinoJson uses converters to convert between the internal types and the public types. For example, when you call `doc.set(42)`, a converter is used to set the `asInteger` field of the `VariantData` object. Similarly, when you call `doc.as<int>()`, a converter reads `asInteger` and casts it to `int`.

These conversions are implemented by specialization of the `Converter` class template, as we saw in the previous chapter. Even if they are called "converters," the built-in ones tend to do more than convert types: they also translate the calls to the private API of `VariantData`. Here is `Converter<bool>` for example:

```
template <>
struct Converter<bool> {
  static void toJson(bool src, JsonVariant dst) {
    VariantData* data = VariantAttorney::getData(dst);
    ResoureseManager* res = VariantAttorney::getResourceManager(dst);
    if (data)
      data->setBoolean(src, res);
  }

  static bool fromJson(JsonVariantConst src) {
    const VariantData* data = VariantAttorney::getData(src);
    return data ? data->asBoolean() : false;
  }

  static bool checkJson(JsonVariantConst src) {
    const VariantData* data = VariantAttorney::getData(src);
    return data && data->isBoolean();
  }
};
```

You should recognize `toJson()`, `fromJson()`, and `checkJson()` from the previous chapter:

- `toJson()` is called when you assign a value to a `JsonVariant` or call `JsonVariant::set()`.

- `fromJson()` is called when you read a value from a `JsonVariant` or call `JsonVariant::as<T>()`.
- `checkJson()` is called when you call `JsonVariant::is<T>()`.

Note how these functions use the `VariantAttorney` to get the `VariantData*` from `JsonVariant` and `JsonVariantConst`. Indeed, `JsonVariant::getData()` and `JsonVariantConst::getData()` are private, so we need a way to access them from the converters. I could have made all the converters friends of `JsonVariant` and `JsonVariantConst`, but I preferred to use the Attorney-Client idiom instead.

The principle of this idiom is that a third party (`Converter<bool>` in this case) cannot talk directly to the client (`JsonVariant`) but has to go through an attorney (`VariantAttorney`). The attorney is a friend of the client, so it can access its private members. Of course, the attorney doesn't reveal all the secrets of its client and only exposes a subset of its interface. In our case, `VariantAttorney` exposes `getData()` and `getResourceManager()`:

```
class VariantAttorney {
 public:
  static ResourceManager* getResourceManager(JsonVariant client) {
    return client.getResourceManager();
  }

  static VariantData* getData(JsonVariant client) {
    return client.getData();
  }
};
```

In reality, `VariantAttorney` is a bit more complex, but this is the general idea.

Nothing prevents you from using the `VariantAttorney` outside the library, so it's not a bulletproof protection, but it should strongly discourage.

In addition to converting between the internal and public types, the converters also perform some validation. For example, the integer converter checks that the value is in the range of the target type and returns 0 if it's not. We talked about that earlier in this chapter.

6.12 Proxies

If you looked at the documentation for `JsonVariant`, you'll see these functions:

```
JsonVariant operator[](const char* key);
JsonVariant operator[](int index);
```

However, if you look at the source code, you'll see that they are actually defined like so:

```
MemberProxy<JsonVariant> operator[](const char* key) const;
ElementProxy<JsonVariant> operator[](const char* key) const;
```

Again, I simplified a bit for simplicity, but the idea is that `JsonVariant` doesn't return a `JsonVariant` when you call `operator[]`, but a proxy object that behaves like a `JsonVariant`. The proxy is either a `MemberProxy` for object members or an `ElementProxy` for array elements.

To understand why we need proxies, let's see what would happen if we didn't have them. Imagine you have this code:

```
JsonDocument doc;
doc["hello"] = "world";
serializeJson(doc, Serial);  // prints {"hello":"world"}
```

To make this code work, `doc["hello"]` would have to create the member "hello" in the document and then return a reference to it. No problem so far. But what if you write this:

```
JsonDocument doc;
const char* world = doc["hello"];  // returns nullptr
serializeJson(doc, Serial);  // prints {"hello":null}
```

Of course! If we need to create a member every time we read it, then the document would have to contain a `null` for each missing member. That's not what we want.

The proxy solves this problem by creating the member only when you assign a value. Apart from that, the proxy behaves like a `JsonVariant`, so you can use it transparently.

In addition to the pointer to its parents, the proxy also contains the key or the index of the member or element. In this case, the parent is a `JsonVariant`, but if you chain the calls, the parent can be another proxy. For example:

```
doc["config"]["network"]["port"] = 2736;
```

This line calls `MemberProxy<MemberProxy<MemberProxy<JsonDocument>>>::operator=(int)` to create the following document:

```
{
  "config": {
    "network": {
       "port": 2736
    }
  }
}
```

`JsonArray`, `JsonDocument`, and `JsonObject` also return proxies, but not `JsonArrayConst`, `JsonObjectConst`, and `JsonVariantConst`. Indeed, we don't need to return proxies for read-only objects; returning a `JsonVariantConst` is enough.

6.13 Deserializers

The deserializers are the classes that parse the input to populate a `JsonDocument`. In ArduinoJson 7, there are two deserializers:

- `JsonDeserializer`, invoked by `deserializeJson()`.
- `MsgPackDeserializer`, invoked by `deserializeMsgPack()`.

On the surface, these two classes are quite simple: they contain member functions like `parseVariant()`, `parseArray()`, and `parseObject()`, which it calls whenever it encounters a value, an array, or an object in the input. The dispatch is done through a `switch` statement on the current character.

6.13.1 Reading from various types

In the previous chapter, we saw you could define a custom reader class to support a new input type. A similar mechanism is used in the deserializers to support various input types. A template class `Reader<TInput>` is used to read the input. It has a specialization for each type of input supported by the deserializers:‘

- `Reader<const char*>` for C strings
- `Reader<std::string>` for standard strings
- `Reader<String>` for Arduino strings
- `Reader<const __FlashStringHelper*>` for Flash strings
- `Reader<Stream>` for Arduino streams
- `Reader<std::istream>` for standard streams

The default implementation of `Reader<TInput>` simply forwards the calls to `TInput`; that's how it supports user-defined reader classes.

A second template class, `BoundedReader<TInput>`, is used when you pass a pointer and a size. It is similar to `Reader<const char*>` and `Reader<const __FlashStringHelper*>`, except that it checks the boundaries of the input.

6.13.2 Reading one character at a time

A valuable 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.

This feature is quite difficult to implement because the deserializer must not eagerly read the next character but only read it when absolutely required. In ArduinoJson 7, `JsonDeserializer` used the `Latch` class to do that. The `Latch` is responsible for reading the input and doesn't read the input until the deserializer is done with the current character and needs the next one.

6.13.3 Nesting limit

Both deserializers use recursive functions. Each time there is a `{` or `[` in the input, `JsonDeserializer` makes a new recursive call to its `parseVariant()` function.

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 could crash the program.

Here is an example:

```
[[[[[[[[[[[[[[666]]]]]]]]]]]]]]
```

The JSON document above contains 15 opening brackets. When the parser reads this document, it enters 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 program's behavior.

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:

```
JsonDocument doc;
deserializeJson(doc, input, DeserializationOption::NestingLimit(15));
int value = doc[0][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.

6.13.4 Escape sequences

The JSON specification defines a list of escape sequences to insert special characters (like newlines and tabs) in strings.

For example:

```
["hello\nworld"]
```

In this document, a newline (`\n`) separates the words `hello` and `world`.

ArduinoJson handles the following escape sequences:

Escape sequence	Meaning
`\"`	Quote (")
`\/`	Forward slash
`\\`	Backslash (\)
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tabulation
`\uXXXX`	Unicode character

When `JsonDeserializer` encounters a Unicode escape sequence (`\uXXXX`), it converts the UTF-16 codepoint (`XXXX`) to UTF-8. It supports surrogate pairs but doesn't raise an error if the codepoint is invalid.

Several classes are involved in this process:

- `EscapeSequence` handles the escape sequences,
- `Codepoint` represents a Unicode codepoint,
- `Utf8` does the Unicode-to-UTF8 conversion.

6.13.5 Filtering

As we saw in the Advanced Techniques chapter, `deserializeJson()` can filter the input document to keep only the values you're interested in. Values excluded by the filter are 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()`.

If you look at the implementation, you'll see that half of this feature is implemented by `DeserializationOption::Filter`, which answers questions like "is an array allowed here?" or "is an object allowed here?". The other half is implemented by `JsonDeserializer`, which checks the filter before parsing each value and skips it if necessary. Skipping is implemented by another set of functions: `skipArray()`, `skipObject()`, and `skipVariant()`.

6.13.6 String-to-float conversion

Converting a string to a float can be tricky when you can't use functions like `atod()` or `strtod()`.

`JsonDeserializer` uses a custom algorithm to convert a string to a float. It consists in extracting the mantissa and the exponent as integers and then combining them to get the final value. In theory, this is not very complicated, but in practice, you must make sure that you support all corner cases and never overflow the integers. You must also choose the right integer type depending on the size of the floating-point type.

The string-to-float conversion is implemented in `parseNumber()`, and the selection of the integer types for the mantissa and the exponent is done in `FloatTraits`.

6.14 Serializers

The serializers are the classes that convert a `JsonDocument` to a textual representation. In ArduinoJson 7, there are three serializers:

- `JsonSerializer`, invoked by `serializeJson()` and `measureJson()`.
- `PrettyJsonSerializer`, invoked by `serializeJsonPretty()` and `measureJsonPretty()`.
- `MsgPackSerializer`, invoked by `serializeMsgPack()` and `measureMsgPack()`.

`measureJson()`, `measureJsonPretty()`, and `measureMsgPack()` simply call the matching `serializeXxx()` function with a dummy writer that does nothing but count the number of bytes.

Performance

`measureJson()` (and two other variants) is costly because it requires a complete serialization. Use it only if you must.

The serializer classes are implemented as visitors (as in the Visitor pattern). `VariantData` "accepts" the visitor and calls the appropriate `visit()` function on the serializer:

- `visit(const ArrayData&)` if it contains an array,
- `visit(const ObjectData&)` if it contains an object,
- etc.

6.14.1 Writing to various types

In the previous chapter, we saw you could define custom writer classes to support new output types. A similar mechanism is used in the serializers to support various output types. A template class `Writer<TOutput>` is used to write the output. It has a specialization for each type of output supported by the serializers:

- `Writer<const char*>` for C strings
- `Writer<std::string>` for standard strings
- `Writer<String>` for Arduino strings

- `Writer<Stream>` for Arduino streams
- `Writer<std::ostream>` for standard streams

The default implementation of `Writer<TOutput>` simply forwards the calls to `TOutput`; that's how it supports user-defined writer classes.

6.14.2 Float-to-string conversion

Even though it looks straightforward, converting a float to a string is amazingly complex. Most programmers don't realize it because they use `printf()` or `sprintf()` to do the conversion, but these functions are not available on Arduino, and even if they were, they would be too big for an embedded system. That is why ArduinoJson implements its own float-to-string conversion 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 changing the number of digits,
3. it doesn't allow changing the exponentiation threshold.

I explained how this algorithm works in the article "Lightweight float to string conversion" in my blog. In a nutshell, it consists in splitting the float into three integers: the integral part, the decimal part, and the exponent. Then, it uses a classic integer-to-string conversion to build the final result. The limitation of 9 decimal places comes from the fact that it stores the decimal part as a 32-bit integer.

The float-to-string conversion algorithm is implemented in `TextFormatter`, but the most interesting part, the splitting of the float into three integers, is done in `FloatParts`.

6.15 Namespaces

All the public symbols of ArduinoJson are defined in the `ArduinoJson::VXXXXXX` namespace, where `XXXXXX` is the version number followed by three characters encoding the configuration of the library for the library.

For example, if I compile version 7.0.0 for an ESP8266 with the default options, the namespace is `ArduinoJson::V700PB2`, but it changes if I change the configuration:

Configuration	Namespace
Default	`ArduinoJson::V700PB2`
`ARDUINOJSON_ENABLE_PROGMEM=0`	`ArduinoJson::V700HB2`
`ARDUINOJSON_USE_LONG_LONG=0`	`ArduinoJson::V700LB2`
`ARDUINOJSON_USE_DOUBLE=0`	`ArduinoJson::V700NB2`
`ARDUINOJSON_ENABLE_NAN=1`	`ArduinoJson::V700PJ2`
`ARDUINOJSON_ENABLE_INFINITY=1`	`ArduinoJson::V700PF2`
`ARDUINOJSON_ENABLE_COMMENTS=1`	`ArduinoJson::V700PD2`
`ARDUINOJSON_DECODE_UNICODE=0`	`ArduinoJson::V700PA2`
`ARDUINOJSON_SLOT_ID_SIZE=4`	`ArduinoJson::V700PB4`

The goal of the inner namespace is to let you use different versions of ArduinoJson in the same program. Of course, it's not an invitation to do so because it will dramatically increase the size of your program.

The inner namespace is an `inline` namespace, so it's implied if you don't specify anything after `ArduinoJson`.

The private symbols of ArduinoJson are defined in the `ArduinoJson::VXXXXX::detail` namespace.

The macros relative to namespaces are defined in `Namespace.hpp`:

- `ARDUINOJSON_VERSION_NAMESPACE` defines the inline namespace (`VXXXXXX`)
- `ARDUINOJSON_BEGIN_PUBLIC_NAMESPACE` and `ARDUINOJSON_END_PUBLIC_NAMESPACE` define the start and end of the public namespace (`ArduinoJson::VXXXXXX`)
- `ARDUINOJSON_BEGIN_PRIVATE_NAMESPACE` and `ARDUINOJSON_END_PRIVATE_NAMESPACE` define the start and end of the private namespace (`ArduinoJson::VXXXXXX::detail`)

6.16 Aggregated header

The single header version of ArduinoJson is generated by a Bash script in `extras/scripts/build-single-header.sh`. This script reads `ArduinoJson.h` in search of `#include` directives and replaces them with the content of the included file. It does that recursively until there is no more `#include` directive.

This script is very slow and probably only works for ArduinoJson but requires very little maintenance. Hopefully, someday, I'll find a dedicated tool to generate the single header.

6.17 Summary

In this chapter, we looked at the library from the inside to understand how it works.

Here are the key points to remember:

- `VariantData`, `ArrayData`, and `ObjectData` are the three classes that store the data.
- `ResourceManager` handles all the memory allocations.
- A `JsonDocument` owns a `VariantData` and a `ResourceManager`.
- A `JsonVariant` points to a `VariantData` and a `ResourceManager`.
- Similarly, a `JsonArray` points to an `ArrayData` and a `ResourceManager`, and a `JsonObject` points to an `ObjectData` and a `ResourceManager`.
- `VariantData` is allocated in a `SlotPool`. More pools are allocated as needed.
- String are allocated and deduplicated in a `StringPool`. There is only one `StringPool` per `JsonDocument`.
- `JsonVariant::operator[]` returns proxies instead of the real thing, so we don't add a `null` when a key is missing.
- Deserializers:
  - There are two deserializer classes: `JsonDeserializer` and `MsgPackDeserializer`.
  - They are recursive but are protected against stack overflow.
  - They use a `Reader` to read the input.
  - `JsonDeserializer` uses `Latch` to read one character at a time.
  - `JsonDeserializer` has a custom string-to-float conversion algorithm.
- Serializers:
  - There are three serializer classes: `JsonSerializer`, `PrettyJsonSerializer`, and `MsgPackSerializer`.
  - They use a `Writer` to write the output.
  - `JsonSerializer` has a custom float-to-string conversion algorithm.

- Even if it looks like everything is defined in the `ArduinoJson` namespace, in reality, it's in the `ArduinoJson::VXXXXXX` namespace, where `XXXXXX` encodes the library's version and configuration.
- Using different versions and configurations of ArduinoJson in the same program is possible but strongly discouraged.

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 Introduction

In this chapter, we'll see how to diagnose and fix the common issues users face when using the library. Some techniques are specific to ArduinoJson, but most apply to any embedded C++ code.

In addition to the ArduinoJson Assistant, `arduinojson.org` also hosts the ArduinoJson Troubleshooter, an online tool that asks you a series of questions to help you fix your program. It asks questions based on your previous answers to narrow down the problem until it finds the issue.

There are many commonalities between this tool and this chapter. The main difference is that they proceed in opposite directions: the Troubleshooter finds the cause from the symptoms, whereas this chapter explains the cause and then describes the possible symptoms. The advantage of the Troubleshooter is that it guides you and doesn't overwhelm you with too much information, but it doesn't encourage you to think by yourself. The advantage of this chapter is that it gives you an overview of all the issues you could encounter and the techniques you could use. Not only does this allow you to avoid these pitfalls, but it also gives you a better understanding of the underlying environment, which helps you write better programs.

In short, the Troubleshooter helps you fix the problems, whereas this chapter allows you to avoid them.

7.2 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 your program may crash.

7.2.1 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 fail 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.

7.2.2 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 causes the program to 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 really caused by the library.

7.2.3 Null string

A common cause of crashes 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
  // The 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, which is the case if `doc["password"]` returns 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 AVR 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
}
```

7.2.4 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 likely to corrupt the heap memory and ultimately cause a crash, but it can also have no incidence. It can also expose a security risk if an attacker exploits how 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 the 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
  JsonDocument 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 you don't need to write 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);
```

7.2.5 Return of stack variable address

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." 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));
```

Code smell

As we saw in these two sections, returning a pointer or a reference from a function is risky. Unfortunately, such functions are prevalent in C++ and cannot be eliminated. When you see one, make sure that the pointer or the reference is still valid when you use it.

7.2.6 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 program's behavior. `strcpy()`, `sprintf()`, and the like are traditional sources of buffer overflow.

Here is a program using ArduinoJson and presenting 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 to change the program's behavior.

We can fix the code above 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 and limit the size, but only `strlcpy()` adds the null-terminator. `strncpy()` is almost as dangerous as `strcpy()`, so make sure you use `strlcpy()`.

7.2.7 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 (such as ESP8266), an exception is raised.
2. On other platforms (such as 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 will likely crash and expose vulnerabilities.

This problem usually occurs when you have large buffers on the stack, for example:

```
char jsonInput[8192];  // <- 8KB on the stack
```

As a general rule, limit the buffer to a quarter of the maximum so that there is plenty of room for other variables and the call stack (function arguments and return addresses). If they are larger, allocate them in the heap instead, possibly using a wrapper class like `String` or `std::string`. Also, remember that ArduinoJson can work directly with streams, so most of the time, you don't even need a buffer.

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?

Such behavior is the sign of a stack overflow. You must reduce the number and the size of variables in the stack.

7.2.8 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 working on professional embedded software. Still, I never used a debugger for Arduino-like projects. I know some tools allow 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 program's operations 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 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 immediately, 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* rather than the *cause*.

To troubleshoot this kind of bug, the best way I know is to remove suspicious lines or 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 the program works fine after doing this simple substitution, then it means there was a problem with this line.

When using this technique, I strongly recommend using 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 you know to be "bad." Git will checkout commits 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 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 improve performance.

7.2.9 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 with a flag that enables code instrumentation. 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 easy to spot:

- functions returning a pointer (like `String::c_str()` or `JsonVariant::as<const char*>()`)
- functions returning a reference
- pointers with a wide scope (i.e., long-lived)
- manual memory management with `malloc()`/`free()` or `new`/`delete`

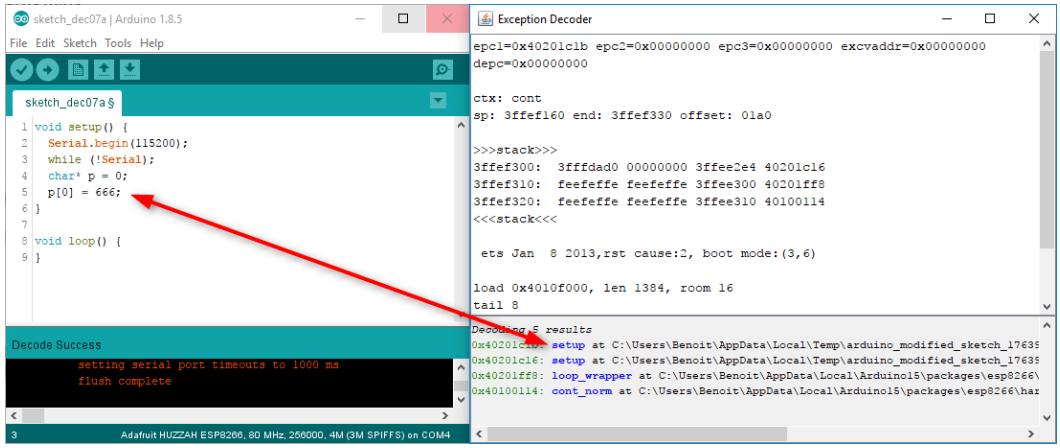
These risky practices should raise a red flag when you review the code, as they are likely to cause trouble. Unfortunately, you cannot eliminate all of them because there are many legitimate usages too.

Making sense of exceptions 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 stack dump. 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`

As I'm writing these lines, EspExceptionDecoder only works with Arduino IDE 1.x, but not with 2.x.


sketch_dec07a | Arduino 1.8.5
File Edit Sketch Tools Help
sketch_dec07a §
1 void setup() {
2 Serial.begin(115200);
3 while (!Serial);
4 char* p = 0;
5 p[0] = 666;
6 }
7
8 void loop() {
9 }
Decode Success
setting serial port timeouts to 1000 ms
flush complete
3
Adafruit HUZZAH ESP8266, 80 MHz, 256000, 4M (3M SPIFFS) on COM4
Exception Decoder
epc1=0x40201c1b epc2=0x00000000 epc3=0x00000000 excvaddr=0x00000000
depc=0x00000000
ctx: cont
sp: 3ffef160 end: 3ffef330 offset: 01a0
>>>stack>>>
3ffef300: 3fffdad0 00000000 3ffee2e4 40201c16
3ffef310: feefeffe feefeffe 3ffee300 40201ff8
3ffef320: feefeffe feefeffe 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
0x40201c16: setup at C:\Users\Benoit\AppData\Local\Temp\arduino_modified_sketch_17635
0x40201ff8: loop_wrapper at C:\Users\Benoit\AppData\Local\Arduino15\packages\esp8266\
0x40100114: cont_norm at C:\Users\Benoit\AppData\Local\Arduino15\packages\esp8266\har

7.3 Deserialization issues

In the chapter Deserialize with ArduinoJson, we saw that `deserializeJson()` returns a `DeserializationError` that can have one of the following values: `Ok`, `EmptyInput`, `IncompleteInput`, `InvalidInput`, `NoMemory`, and `TooDeep`.

In this section, we'll see when these errors occur and how to fix them.

7.3.1 `EmptyInput`

`DeserializationError::EmptyInput` means that the input was empty or contained only spaces or comments.

Reason 1: Input stream timed out

`EmptyInput` often means that a timeout occurred before the program got a chance to read the document. This usually happens when reading from a serial port, waiting for a new message.

To fix this problem, you can wait until some data is available before calling `deserializeJson()`, like so:

```
// wait for data to be available
while (Serial.available() == 0)
  delay(100);

// read input
deserializeJson(doc, Serial1);
```

After adding this waiting loop, if you still get `EmptyInput`, it means that the input is made of spaces or line breaks. Usually, these characters follow the JSON document in the input stream. For example, the Arduino Serial Monitor appends CRLF (`\r\n`) when you click "Send."

The best workaround is to flush the remaining blank characters after calling `deserializeJson()`. You can do so with another loop:

```
// wait for data to be available
while (Serial.available() == 0)
  delay(100);

// read input
deserializeJson(doc, Serial1);

// flush blank characters
while (isspace(Serial.peek()))
  Serial.read();
```

Reason 2: HTTP redirect

An HTTP server may return a status code 301 or 302 to indicate a redirection. In this case, the response body is empty, so calling `deserializeJson()` will result in an `EmptyInput` error.

To fix this problem, you must update the URL or handle the redirection in your program when the server returns 301 or 302. If you use an HTTP library, configure it to follow redirections.

7.3.2 `IncompleteInput`

ArduinoJson returns `DeserializationError::IncompleteInput` when the beginning of the document is correct, but the end is missing.

Reason 1: Input buffer is too small

Typically, `deserializeJson()` returns `IncompleteInput` when the program uses a buffer 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 `deserializeJson()` returns `IncompleteInput`.

I used a char array in this example, but the same problem would happen with a `String`. If the heap is exhausted or fragmented, `String` cannot reallocate a sufficiently large buffer and drops the end of the input.

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);
```

Reason 2: connection dropped

`IncompleteInput` also happens when the connection that carries the JSON document drops. This drop can be due to an unreliable connection or a slow read pace. There is not much we can do about the quality of the connection at the software level, so let's see how we can deal with the speed problem.

`deserializeJson()` reads bytes one-by-one, which can be pretty slow with some implementations of `Stream`. If the receiver reads too slowly, it might drop the connection and miss the end of the message.

To improve the reading speed, we must insert a buffer between the stream and `deserializeJson()`. We can do that with the `ReadBufferingStream` class from the StreamUtils library.

```
ReadBufferingStream bufferedStream(stream, 64);
deserializeJson(doc, bufferedStream);
```

See the Advanced Techniques chapter for more information.

Reason 3: Input stream timed out

`IncompleteInput` can also mean 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.setTimout(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 transmission quality; for example, you could reduce the speed to improve the error ratio.

7.3.3 `InvalidInput`

`deserializeJson()` returns `DeserializationError::InvalidInput` when the input contains something 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 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

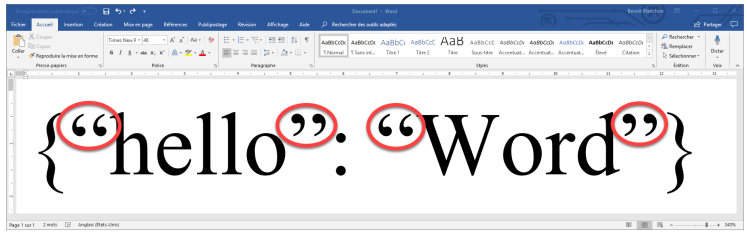
`InvalidInput` can be caused by one or several incorrect characters in the input. For example, it can happen if you use a serial connection between two boards. Indeed, the serial connection often inserts errors in the transmission.

If that happens to you, you can lower 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: Invalid JSON

Of course, `deserializeJson()` also returns `InvalidInput` if the input is not a valid JSON document. Some common mistakes are:

1. Incorrect quotation marks: JSON uses straight quotation marks (`"`) and doesn't allow curly quotation marks (`“` and `”`).
2. Trailing commas: JSON doesn't allow trailing commas in arrays and objects.
3. Hexadecimal numbers: JSON doesn't only allow decimal numbers.



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 problem is that the program doesn't skip the HTTP headers before parsing the body. To fix this bug, 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 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 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 `ï»¿`.

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>
```

7.3.4 NoMemory

`deserializeJson()` returns `DeserializationError::NoMemory` when it cannot allocate memory for the `JsonDocument`. In other words, it means that the heap is full.

You need to make some room in the heap and fight heap fragmentation. I recommend that you start by removing as many `String`s 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 Advanced Technique chapter.

7.3.5 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 argument to `deserializeJson()`:

```
deserializeJson(doc, input, DeserializationOption::NestingLimit(15));
```

Alternatively, you can change the default nesting limit by defining the macro `ARDUINOJSON_DEFAULT_NESTING_LIMIT`:

```
#define ARDUINOJSON_DEFAULT_NESTING_LIMIT 15
#include <ArduinoJson.h>
```

As we saw, the ArduinoJson Assistant warns you about this problem and includes the extra argument in step 3.

Step 3: Program

Serial Flash strings

Copy

```
// String input;

JsonDocument doc;

DeserializationError error = deserializeJson(doc, input, DeserializationOption::NestingLimit(15));

if (error) {
  Serial.print("deserializeJson() failed: ");
  Serial.println(error.c_str());
  return;
}

int root_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0 = doc[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]; // 666
```

See also Deserialization Tutorial `deserializeJson()`

Previous

7.4 Serialization issues

In this section, we'll see the most common problems you might get when serializing a JSON document.

7.4.1 The JSON document is incomplete

If the generated JSON document misses some parts, it's because the `JsonDocument` failed to allocate enough memory.

For example, suppose you want to generate the following:

```
{
  "firstname": "Max",
  "lastname": "Power"
}
```

If, instead, you get the following output:

```
{
  "firstname": "Max"
}
```

It means memory allocation for the member `lastname` failed. There is no easy solution here: you need to make some room on the heap, or at least you must reduce the fragmentation. Again, start by hunting the `String` instances because they are the most likely to cause fragmentation.

By the way, if you are afraid to send or save an incomplete JSON document, you check `JsonDocumet::overflowed()`; it returns `true` if any previous allocation failed.

7.4.2 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();
}

JsonDocument doc;
doc["pin"] = pinName(0);
serializeJson(doc, Serial);
```

Naturally, the author of this program expects the following output:

```
{"pin":"D0"}
```

Instead, she is very likely to get something like that:

```
{"pin":"sÜd4xaÜY9ËåQ¥º;"}
```

The JSON document contains garbage because `obj["pin"]` stores a pointer to a destructed object. Indeed, the temporary `String` declared in `pinName()` dies as soon as the function exits. 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.

7.4.3 The serialization is slow

`serializeJson()` writes most of the document one character at a time, which can be pretty slow with unbuffered streams.

The solution is to insert a buffer between the stream and `serializeJson()`. We can do that with the `BufferingPrint` class from the StreamUtils library.

```
BufferingPrint bufferedStream(stream, 64);
serializeJson(doc, bufferedStream);
bufferedStream.flush();
```

See the Advanced Techniques chapter for more information.

7.5 Common error messages

In this section, we'll see the most frequent compilation errors and how to fix them.

7.5.1 Invalid conversion from const char* to char*

This error occurs when you try to assign a `const char*` to a `char*`:

This error occurs when you try to store a pointer of type `const char*` into a variable of type `char*`. For example:

```
char* name = doc["name"];
```

Indeed, JsonVariant returns a `const char*`, not a `char*`. You must change the type of the pointer like so:

```
const char* name = doc["name"];
```

7.5.2 Invalid conversion from const char* to int

Let's say you have to deserialize the following JSON document:

```
{
  "rooms": [
    {
      "name": "kitchen",
      "ip": "192.168.1.23"
    },
    {
      "name": "garage",
      "ip": "192.168.1.35"
    }
  ]
}
```

If you write the following program:

```
deserializeJson(doc, input);
JsonArray rooms = doc["rooms"];

const char* ip = rooms["kitchen"]["ip"];
```

You'll get the following compilation error:

```
invalid conversion from 'const char*' to 'size_t {aka unsigned int}'...
```

`rooms` 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[]`:

```
~~const char* ip = rooms["kitchen"]["ip"];~~
const char* ip = rooms[0]["ip"];
```

Now, if you need to find the room named "kitchen," you need to loop and check each room one by one:

```
for (JsonObject room : rooms) {
  if (room["name"] == "kitchen") {
    const char* ip = room["ip"];
    // ...
  }
}
```

7.5.3 No match for operator[]

This error often 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<const char*>());
}
```

7.5.4 Ambiguous overload for operator=

You can rely on implicit casts most of the time, 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>();
```

7.5.5 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
```

7.6 Asking for help

If none of the directions in this chapter helps, you should try the ArduinoJson Troubleshooter. It covers many more cases, so it will likely help you.

If the Troubleshooter doesn't help, I recommend opening a new issue on GitHub.

When you write your issue, please take the time to write a good description. Providing the right amount of information is essential: if you give too little (for example, just an error message without any context), I'll have to ask for more, but 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` or Wokwi and share the link in the issue description. Writing an MCVE seems cumbersome, but it guarantees that the recipient (me, most likely) understands the problem quickly. Moreover, we often find a solution 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. Also, remember Jon Skeet's golden rule, "Imagine You're Trying To Answer The Question", from its classic article Writing the perfect question.

GitHub issues for ArduinoJson usually get answered in less than 24 hours. Of course, I treat bugs with a higher priority than questions, especially when the answer is in the documentation.

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
- ArduinoJson's revision list
- GitHub releases page

7.7 Summary

In this chapter, we saw how to diagnose and solve the most common problems you may have with ArduinoJson.

Here are the key points to remember:

- A bug is more likely 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`.
- "Invalid conversion from `const char*` to `char*`" means you must use `const char*` instead of `char*`.
- "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>()`
- 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 apply what we learned in this book.

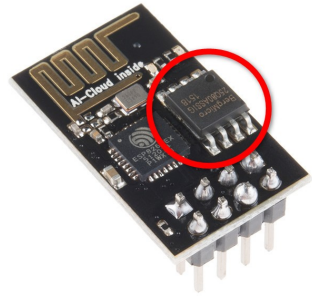
Chapter 8
Case Studies

I'm not a great programmer; I'm just a good programmer with great habits.
– Kent Beck

8.1 JSON Configuration File

8.1.1 Presentation

For our first case study, we'll consider a program that stores its configuration in a file. The program loads a global configuration object at startup and saves it after each modification.



In our example, we'll consider the file system LittleFS, but you can quickly adapt the code to any other. LittleFS 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 `ConfigFile` folder of the zip file provided with the book.

SPIFFS

You can easily adapt the code to use SPIFFS: simply replace all instances of `LittleFS` with `SPIFFS`.

8.1.2 The JSON document

Here is the layout of the configuration file that we're going to use in this case study:

```
{
  "access_points": [
    {
      "ssid": "SSID1",
      "passphrase": "PASSPHRASE1"
    },
    {
      "ssid": "SSID2",
      "passphrase": "PASSPHRASE2"
    }
```

```
  ],
  "server": {
    "host": "www.example.com",
    "path": "/resource",
    "username": "admin",
    "password": "secret"
  }
}
```

This configuration contains two main parts:

1. a list of access points,
2. a web service.

We assume the following constraints:

- there can be up to 4 access points,
- an SSID has up to 31 characters,
- a passphrase has up to 63 characters,
- each server parameter has up to 31 characters.

8.1.3 The configuration class

The best way to store the configuration in memory is to create a structure containing all the parameters.

We'll mirror the hierarchy of the JSON document in the configuration classes. While it's not mandatory, it dramatically simplifies the conversion 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 access points and a `ServerConfig` to store the web service configuration.

8.1.4 Converters

In the Advanced Techniques chapter, we saw that ArduinoJson supports user-defined types through "custom converters." We'll use this feature to insert (and extract) a `Config` instance into (from) a `JsonDocument`.

To save a `Config` into a `JsonDocument`, we must define the following function:

```
void convertToJson(const Config &src, JsonVariant dst);
```

This function must somehow set the `JsonVariant` from the `Config`. Similarly, to load a `Config` object from a `JsonDocument`, we must create the following function, which performs the reverse operation:

```
void convertFromJson(JsonVariantConst src, Config &dst);
```

Since `Config` contains `ApConfig` and `ServerConfig`, we must also create converters for them. We'll start with `ApConfig`.

Converting ApConfig

Here is the JSON object corresponding to one `ApConfig`.

```
{
  "ssid": "SSID1",
  "passphrase": "PASSPHRASE1"
}
```

Because each field of `ApConfig` maps directly to the member of the JSON object, the conversion code is really straightforward. First, let's see the function that converts an `ApConfig` into a JSON object:

```
void convertToJson(const ApConfig &src, JsonVariant dst) {
  dst["ssid"] = src.ssid;
  dst["passphrase"] = src.passphrase;
}
```

As you can see, we set the members of the JSON object from the fields of `ApConfig`. Now, let's see the function that loads an `ApConfig` from a JSON object:

```
void convertFromJson(JsonVariantConst src, ApConfig &dst) {
  strlcpy(dst.ssid, src["ssid"] | "", sizeof(dst.ssid));
  strlcpy(dst.passphrase, src["passphrase"] | "", sizeof(dst.passphrase));
}
```

As you can see, we apply the technique we learned in the chapter Deserializing with ArduinoJson:

1. We call `strlcpy()` instead of `strcpy()` because it takes an additional parameter that prevents buffer overruns.
2. We provide a non-null default value with the operator `|` to avoid the undefined behavior in `strlcpy()`

This way, we are sure that our program is safe, even in the presence of a rogue configuration file. Note that the default value doesn't have to be an empty string; you can choose something more meaningful.

Now, let's see the next structure, `ServerConfig`.

Converting ServerConfig

Here are the two conversion functions for `ServerConfig`:

```
void convertToJson(const ServerConfig &src, JsonVariant dst) {
  dst["username"] = src.username;
  dst["password"] = src.password;
  dst["host"] = src.host;
  dst["path"] = src.path;
}

void convertFromJson(JsonVariantConst src, ServerConfig &dst) {
  strlcpy(dst.username, src["username"] | "", sizeof(dst.username));
  strlcpy(dst.password, src["password"] | "", sizeof(dst.password));
  strlcpy(dst.host, src["host"] | "", sizeof(dst.host));
  strlcpy(dst.path, src["path"] | "", sizeof(dst.path));
}
```

There is nothing new here: it's the same procedure we followed for `ApConfig`.

I take this opportunity to repeat that the names are imposed by the library; if you rename these functions, ArduinoJson will not find them. Also, these functions must belong to the namespace of the target type, which is the global namespace in this case. We talked about that in the Advanced Techniques chapter, remember?

Converting Config

The `Config` class is a little more complex because it contains an array of `ApConfig`. Here is the first converter:

```
void convertToJson(const Config &src, JsonVariant dst) {
  dst["server"] = src.server;

  JsonArray aps = dst["access_points"].to<JsonArray>();
  for (int i = 0; i < src.accessPoints; i++)
    aps.add(src.accessPoint[i]);
}
```

In the first line, we insert the `ServerConfig` in the "server" member. This assignment triggers the custom converters feature, which calls the `convertToJson()` function we wrote previously.

Then, we create an array and insert each `ApConfig` one by one. Here, too, the custom converters feature is at play: it calls `convertToJson()` to populate the JSON objects from each `ApConfig`.

Let's see the other converter:

```
void convertFromJson(JsonVariantConst src, Config &dst) {
  dst.server = src["server"];

  JsonArrayConst aps = src["access_points"];
  dst.accessPoints = 0;
  for (JsonVariantConst ap : aps) {
    dst.accessPoint[dst.accessPoints++] = ap;
    if (dst.accessPoints >= Config::maxAccessPoints)
      break;
  }
}
```

The first line extracts the `ServerConfig` using our custom converter. Then, the function resets the size of the array and appends the `ApConfig` one after the other. Of course, it ensures the array doesn't overflow, protecting the program from a rogue input.

That's all for the mapping of data structures to JSON. Now, let's see how we can save the JSON document into a file.

8.1.5 Saving the configuration to a file

In the previous section, we wrote a `convertToJson()` that fills a JSON object from a `Config` class. Now, we just need to pass a `Config` instance to `JsonDocument::set()` to populate the entire document.

Here is the function that saves a `Config` into a file:

```
void saveConfigFile(const char *filename, const Config &config) {
  File file = LittleFS.open(filename, "w");
  JsonDocument doc;
  doc.set(config);
  serializeJson(doc, file);
}
```

The first line opens the file for writing. The second allocates the `JsonDocument`, and the third populates it. The last line serializes the document into the file.

No need to close the file: the destructor of `File` takes care of it. Yes, it's another instance of the RAII idiom.

8.1.6 Reading the configuration from a file

The file reading function is very similar to the previous one, except it calls `JsonDocument::as<T>()` instead of `JsonDocument::set()`.

```
Config loadConfigFile(const char *filename) {
  File file = LittleFS.open(filename, "r");
  JsonDocument doc;
  deserializeJson(doc, file);
  return doc.as<Config>();
}
```

The first line opens the file in reading mode. The second and third lines deserialize the content of the file. The last line extracts the `Config` from the `JsonDocument`, leveraging our custom converter.

If you open the sample files, you'll see that I return a boolean to indicate the success of the operation; the `Config` object is passed by reference. It's less elegant than the code above, but we cannot do better without throwing exceptions.

8.1.7 Conclusion

I also hope this case study convinced you that "user-defined structures + converters" is the pattern of choice to store a configuration. If you look at the source code in the `ConfigFile` folder of the zip file, you'll see that this pattern allows a clear separation of concerns:

- The data structures are in `Config.h`, independent of ArduinoJson.
- The JSON converters are isolated in `ConfigJson.h` and `ConfigJson.cpp`
- The file-related code is confined in `ConfigFile.ino`.

As you saw, it's an elegant solution to a not-so-simple problem, and it should scale nicely when the complexity of the data structure increases. In my opinion, the only drawback of this pattern is that it couples the data structure with the layout of the JSON document. This coupling might be problematic if the JSON layout evolves, especially if you must support the old file formats. Nothing impossible, though, but that's something you should keep in mind.

Creating the `Config` data structure represents 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. A `JsonDocument` allows objects to contain several types of values (variants). This flexibility isn't free. Compared to a custom data structure, it adds overhead in speed, memory consumption, and program size.

2. 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.

3. By restricting the use of ArduinoJson to the serialization functions, you decouple the rest of your code from the library. First, it lets you 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 an Arduino 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 you already read the GitHub example, where you learned how to deserialize a JSON document from an HTTP response, and the Adafruit IO example, where we saw how to perform an HTTP request without an HTTP library.

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 five 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." As I explained, this forbids the server to send the response in multiple pieces, which significantly simplifies our program.

The response should look like this:

```
HTTP/1.0 200 OK
Server: openresty
Date: Mon, 27 Nov 2023 13:43:00 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 15809
Connection: close

{"cod":"200","message":0,"cnt":40,"list":[{"dt":1701097200,"main":{...
```

8.2.3 The JSON response

The body of the response is a minified JSON document of 16 KB, which looks like this:

```
{
  "cod": "200",
  "message": 0,
  "cnt": 40,
  "list": [
    {
      "dt": 1701097200,
      "main": {
        "temp": 7.27,
        "feels_like": 4.36,
        "temp_min": 5.79,
        "temp_max": 7.27,
        "pressure": 996,
        "sea_level": 996,
        "grnd_level": 995,
        "humidity": 90,
        "temp_kf": 1.48
      },
```

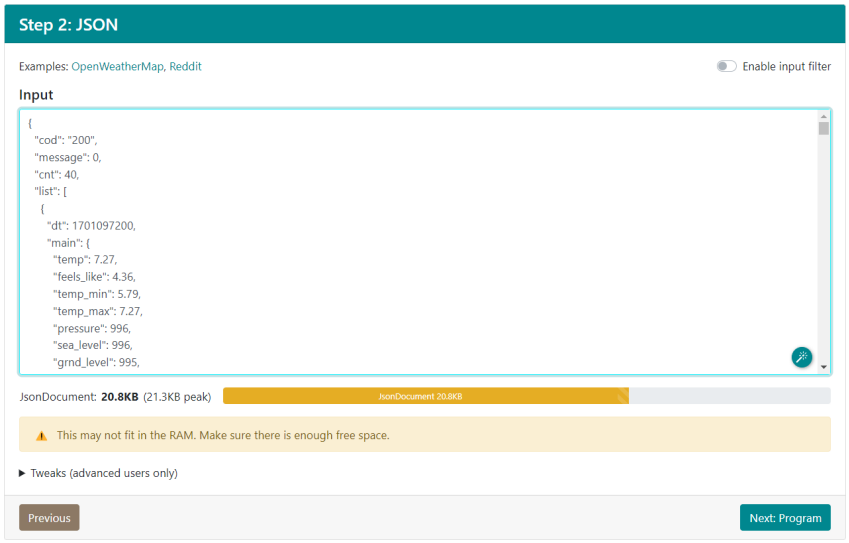
```
      "weather": [
        {
          "id": 500,
          "main": "Rain",
          "description": "light rain",
          "icon": "10d"
        }
      ],
      "clouds": {
        "all": 100
      },
      "wind": {
        "speed": 4.59,
        "deg": 350,
        "gust": 7.09
      },
      "visibility": 10000,
      "pop": 0.96,
      "rain": {
        "3h": 0.62
      },
      "sys": {
        "pod": "d"
      },
      "dt_txt": "2023-11-27 15:00:00"
    },
    ...
  ],
  "city": {
    "id": 2643743,
    "name": "London",
    "coord": {
      "lat": 51.5085,
      "lon": -0.1257
    },
    "country": "GB",
    "population": 1000000,
    "timezone": 0,
    "sunrise": 1701070656,
```

```
    "sunset": 1701100727
  }
}
```

In the sample above, I preserved the hierarchical structure but 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, the `JsonDocument` for this huge response consumes 20.8 KB of RAM. In theory, it should fit in the 32 KB of the Arduino MKR1000, but in practice, it would leave no margin for other things that need RAM, such as the WiFi library.



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:

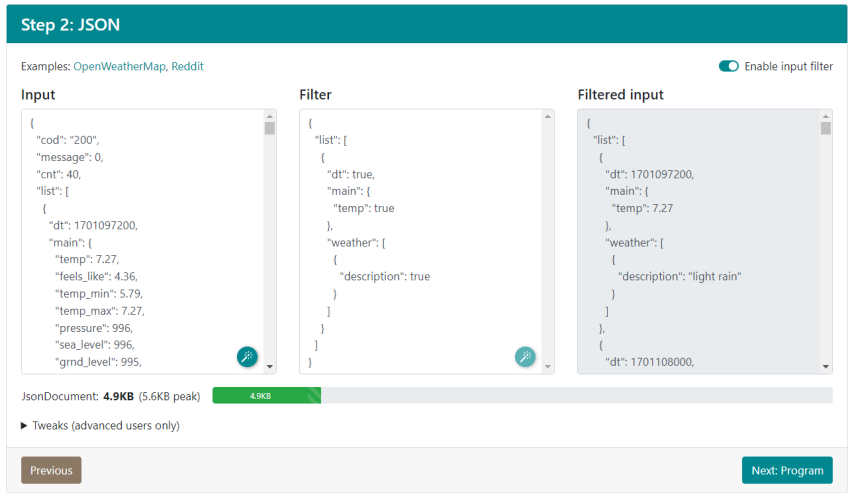
- `"dt"`, which contains the timestamp for the forecast,
- `"temp"`, which contains the temperature in degrees,
- `"description"`, which contains a textual description of the weather condition.

After applying the filter, the JSON document should look like this:

```
{
  "list": [
    {
      "dt": 1627290000,
      "main": {
        "temp": 17.97
      },
      "weather": [
        {
          "description": "overcast clouds"
        }
      ]
    },
    {
      "dt": 1627300800,
      "main": {
        "temp": 19.78
      },
      "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 only 4.9 KB of RAM, which largely fits in the MKR1000. We could reduce memory consumption further by applying the "deserialize in chunks" method; we'll see that in the next case study.



8.2.5 The filter document

Let's look at the filter itself. As we saw in the Advanced Techniques chapter, we must 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 has 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`:

```
JsonDocument 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/forecast?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
JsonDocument doc;
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<const char *>());
  Serial.print(", ");
  Serial.print(forecast["main"]["temp"].as<float>());
  Serial.println(" °C");
}
```

8.2.7 Summary

This second case study showed how to reduce 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 acts as a filter.
- Insert `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 memory consumption: deserialize the input in chunks.

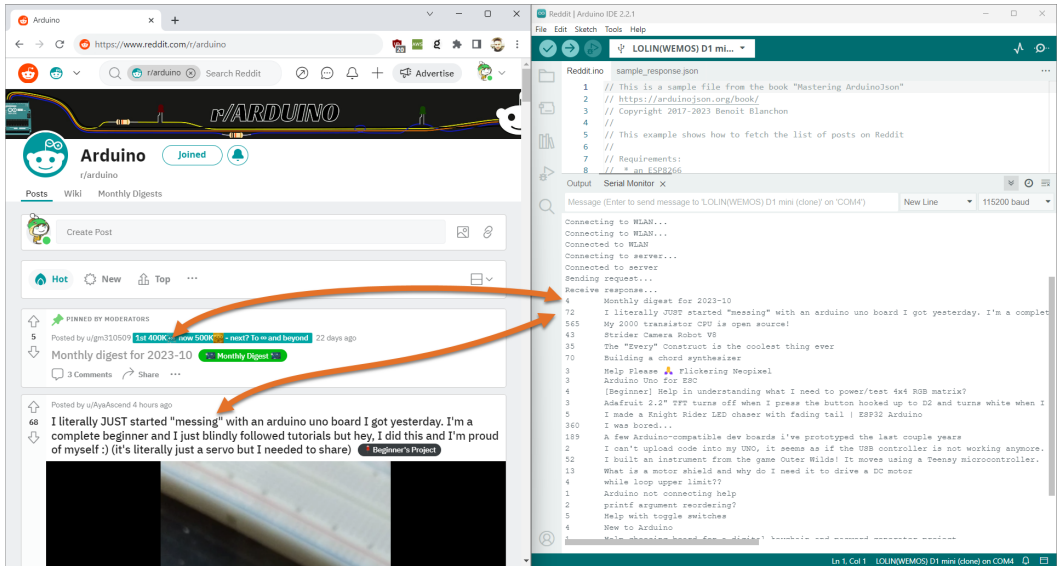
8.3 Reddit on ESP8266

8.3.1 Presentation

We'll create a program that interacts with Reddit, a social network where users share links with other members. Reddit has been around since 2005 and hosts 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 posts 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:



On the left, you see Reddit in my browser; on the right, you see the Arduino Serial Monitor. The program prints the score and the title for the first page of results. The scores change very quickly, which is why they are not the same on both sides.

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. Indeed, each Reddit post only requires between 5 and 16 KB of RAM, but the full document requires more than 200 KB, so reading all posts at once would be impossible.

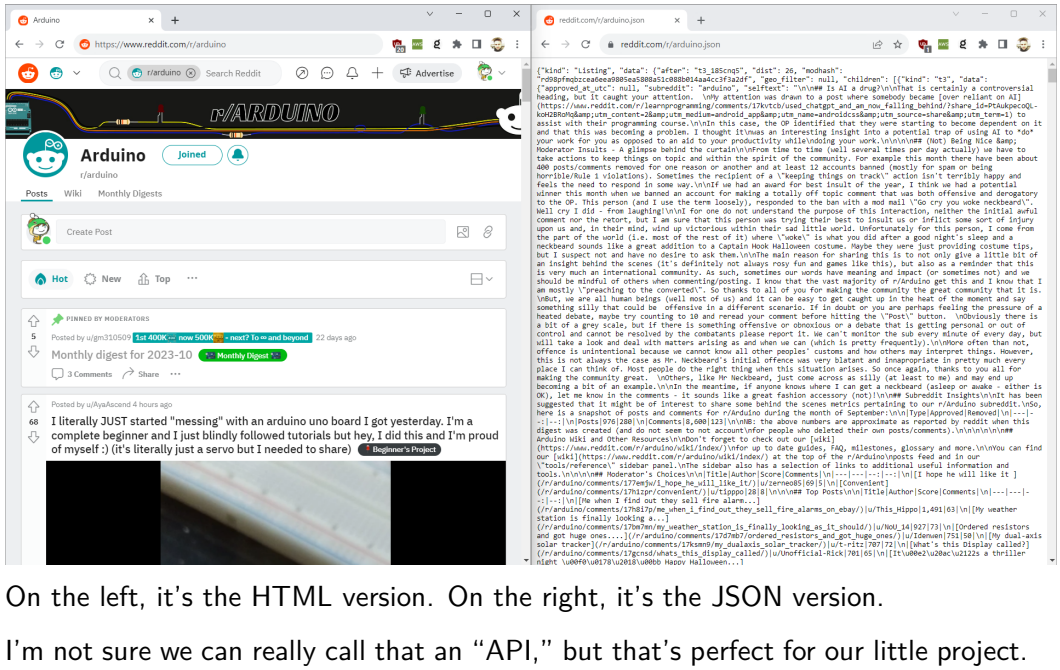
We'll use an ESP8266, like in the GitHub example. This time, however, we'll not use the library ESP8266HTTPClient; instead, we'll work directly with `WiFiClientSecure`. We'll not check the SSL certificate of the server because we don't transmit sensitive information.

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 virtually any URL to get the page in JSON format. 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 same 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": {
    "dist": 26,
    "children": [
      {
        "kind": "t3",
        "data": {
          "title": "Monthly digest for 2023-10",
          "score": 6,
          "author": "gm310509",
          "num_comments": 3,
          ...
        }
      },
      {
        "kind": "t3",
        "data": {
          "title": "I literally JUST started \"messing\" with an arduino uno
↪  board I got yesterday. I'm a complete beginner and I just blindly
↪  followed tutorials but hey, I did this and I'm proud of myself :) (it's
↪  literally just a servo but I needed to share)",
          "author": "AyaAscend",
          "num_comments": 10,
          ...
        }
      },
      ...
    ]
  }
}
```

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 26 to 2. Then, I reduced the number of values in each post from a hundred to four.

For this project, we are only interested in the `data.children` array. This array contains 26 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

As I said, we'll parse each post one by one, so 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 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()`. The first locates the "children" key, and the second jumps to the beginning of the array. One call is usually enough, but I wanted to show you how to write a program that is resilient to changes in the formatting of the input. Indeed, there could be spaces or line breaks between the key, the colon, and the opening bracket. By calling `Stream::find()` twice, we make the program more robust.

Once 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 must call `deserializeJson()` again to deserialize the next post. If it's a closing bracket, it means 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, `false` otherwise. Therefore, we can use the return value as a stop condition for the loop.

Here is the code of the loop:

```
do {
  JsonDocument doc;
  deserializeJson(doc, client);
  // ...extract values from the document...
} while (client.findUntil(",", "]"));
```

8.3.5 Sending the request

Unlike OpenWeatherMap, Reddit forbids plain HTTP requests; it only accepts HTTPS. So, instead of `WiFiClient` and port 80, we must use `WiFiClientSecure` and port 443.

```
WiFiClientSecure client;
client.connect("www.reddit.com", 443);
```

Apart 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.

8.3.6 Assembling the puzzle

I think we covered all the important features of this case study. Let's put everything together:

```
#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("[");

  do {
    // Deserialize the next post
    JsonDocument doc;
    deserializeJson(doc, client);

    // Print score and title
    Serial.print(doc["data"]["score"].as<int>());
    Serial.print('\t');
    Serial.println(doc["data"]["title"].as<const 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.

You'll see that I also added a filter because, some days, the response is so large that even one post doesn't fit in the `JsonDocument`. Indeed, I've seen posts that required more than 50 KB because of extremely long strings in the `selftext` member. Keeping only the title and the score allows us to reduce the size of the `JsonDocument` to a few hundred bytes.

8.3.7 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.

In the next case study, we'll see how to create a reusable REST client.

8.4 RESTful client

8.4.1 Presentation

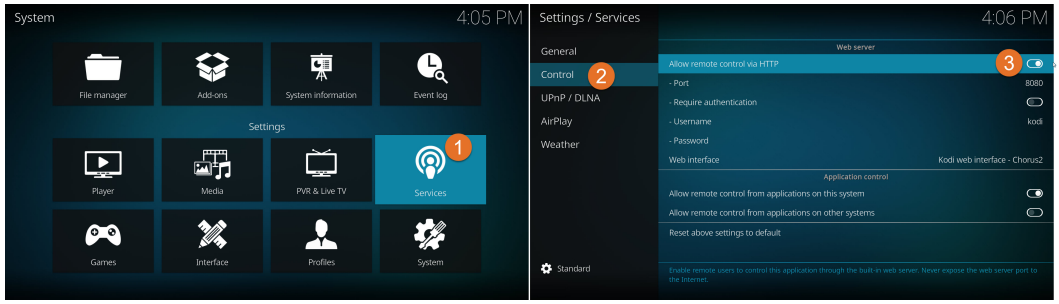
For our fourth case study, we'll create a remote controller for Kodi. Kodi (formerly known as XBMC) is a software media player: it allows playing videos and music on a computer. It has a full-screen GUI designed to be controlled with a TV remote. Kodi is the most popular software to create an HTPC (Home Theater Personal Computer).

Kodi can also be controlled remotely via a network connection; we'll use this feature to control it from 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.

This case study illustrates how easy it is to build a RESTful client with ArduinoJson. We'll create a reusable `RestClient` class that you can use with most RESTful APIs. A RESTful API is an application program interface (API) that uses HTTP methods to manipulate and transfer data. I don't want to go into the details of RESTful APIs, but if you want to learn more, I recommend the book "RESTful Web APIs" by Leonard Richardson and Mike Amundsen.

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 must be careful not to waste it.

The code for this case study is in the `KodiRemote` folder of the zip file you received with this book. To run this program, you need to run an instance of Kodi and check "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. The first one indicates the version of the protocol, and the second one is used to match the request with the response, which is not relevant in our case.

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 a JSON object is very common.

If the 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 reusable RESTful client

In order to help us send JSON-RPC requests and receive JSON-RPC responses, we'll create a reusable `RestClient` class that can send HTTP requests with a JSON body and receive HTTP responses with a JSON body.

```
class RestClient {
public:
  RestClient(const char *host, short port) : host_(host), port_(port) {}

  // Sends a GET request
  int get(const char *path, JsonDocument &doc) {
    return request("GET", path, doc);
  }

  // Sends a POST request
```

```
  int post(const char *path, JsonDocument &doc) {
    return request("POST", path, doc);
  }

private:
  // Sends a request and reads the response.
  // The JsonDocument is sent in the request then filled with the response.
  // Returns the HTTP status code or -1 if the connection failed.
  int request(const char *method, const char *path, JsonDocument &);

  EthernetClient client_;
  const char *host_;
  short port_;
};
```

As you can see, this class offers two functions: `get()` and `post()`. The first sends a GET request, and the second sends a POST request. Both take a path and a `JsonDocument` as parameters. The path is the URL of the resource to access, and the `JsonDocument` is the body of the request/response.

All the magic happens in the `request()` function, which sends the HTTP request and reads the HTTP response. Let's see how it works.

Step 1: (re)connect to the server

The first thing `request()` does is check if the connection is already established. If not, it tries to connect to the server:

```
if (!client_.connected() && !client_.connect(host_, port_)) {
  Serial.print(F("Failed to connect to "));
  Serial.print(host_);
  Serial.print(':');
  Serial.println(port_);
  return -1;
}
```

Nothing special here. This is straight out of the examples of the Ethernet library.

Step 2: send the HTTP request line

The request line is the first line of the HTTP request. It contains the method, the path, and the HTTP version. For example, here is the request line for our POST request to Kodi's JSON-RPC endpoint:

```
POST /jsonrpc HTTP/1.0
```

The `request()` function sends this line with the following code:

```
client_.print(method);
client_.print(' ');
client_.print(path);
client_.println(F(" HTTP/1.0"));
```

Step 3: send the HTTP headers

The HTTP headers are a list of key-value pairs that describe the request. They immediately follow the request line. For example, here are the three headers we'll send immediately after the request line:

```
POST /jsonrpc HTTP/1.0
Content-Type: application/json
Connection: keep-alive
Content-Length: 42
```

The first header indicates that the request's body is in JSON format. The second header asks to keep the connection alive after the request (the default is to close immediately after the response). The third header indicates the length of the body.

The `request()` function sends these headers with the following code:

```
client_.println("Content-Type: application/json");
client_.println("Connection: keep-alive");
client_.print("Content-Length: ");
client_.println(measureJson(doc));
client_.println();
```

Notice the last call to `println()`: it sends an empty line to mark the end of the headers.

Step 4: send the HTTP body

The request body contains the JSON document. No surprise here; we have done that several times already:

```
serializeJson(doc, client_);
```

The request is now complete, and we can wait for the response.

Step 5: read the response status line

An HTTP response is very similar to an HTTP request. It starts with a status line, followed by a list of headers and a body. The status line contains the HTTP version, the status code, and the status message. For example, here is a successful response:

```
HTTP/1.0 200 OK
```

We receive this line with the following code:

```
char statusLine[32] = {0};
client_.readBytesUntil('\r', statusLine, sizeof(statusLine));
```

Calling `readBytesUntil()` with `'\r'` as the delimiter allows us to stop reading when we reach the end of the line. The `sizeof()` operator is used to prevent buffer overflows.

The most important part is the status code (`200` in this case). It indicates that the request was successful. We convert this status code to an integer with the following code:

```
int statusCode = atoi(statusLine + 9);
```

The `atoi()` function converts a string to an integer. The curious `+9` on the pointer allows us to skip the first nine characters so that `atoi()` only sees the status code. By design, `atoi()` stops at the first non-digit character, so we don't need to worry about the rest of the line.

Step 6: read the HTTP headers

As the HTTP request, the HTTP response contains a list of headers. For example, here are the headers returned by Kodi after the status line:

```
HTTP/1.0 200 OK
Connection: Keep-Alive
Content-Length: 38
Content-Type: application/json
Cache-Control: private, max-age=0, no-cache
Accept-Ranges: none
Date: Thu, 30 Nov 2023 20:02:44 GMT
```

We don't need to read these headers for our application, so we can skip them with the following code:

```
client_.find("\r\n\r\n");
```

This code searches for the end of the headers, which is indicated by an empty line. We have already seen this technique in the tutorial.

Step 7: read the HTTP body

The HTTP body contains the JSON document. We read it with the following code:

```
deserializeJson(doc, client_);
```

This should be familiar now.

8.4.5 Sending 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:

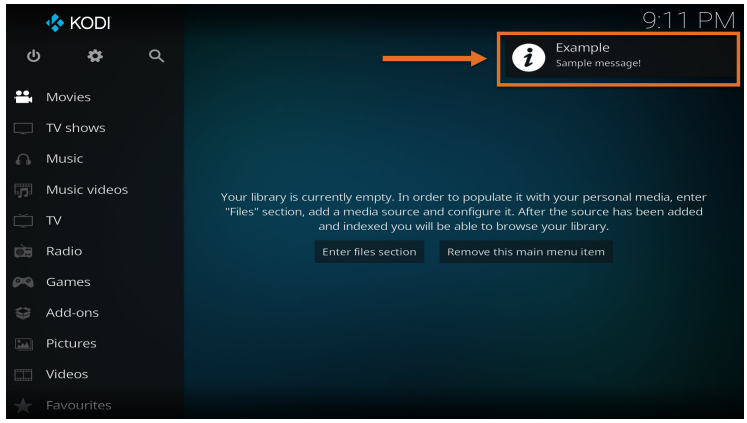
```
{
  "jsonrpc": "2.0",
  "result": "OK",
  "id": 1
}
```

Let's use our new `RestClient` class to send this request:

```
RestClient client(host, port);

JsonDocument doc;
doc["jsonrpc"] = "2.0";
doc["method"] = "GUI.ShowNotification";
doc["params"]["title"] = title;
doc["params"]["message"] = message;
doc["id"] = 1;

_client.post("/jsonrpc", doc);

bool success = doc["result"] == "OK";
```

As you can see, we start by creating a `RestClient` object. Then, we create a `JsonDocument` and fill it out with the request. Finally, we send the request with the `post()` function.

Here is what the notification looks like in Kodi:



8.4.6 Reading Kodi's version

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 of 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": 20,
      "minor": 2,
      "revision": "20230629-5f418d0b13",

      "tag": "stable"
    }
  }

}
```

Let's use our framework again:

```
RestClient client(host, port);

JsonDocument doc;
doc["jsonrpc"] = "2.0";
doc["id"] = 1;
doc["method"] = "Application.GetProperties";
doc["params"]["properties"].add("name");
doc["params"]["properties"].add("version");

_client.post("/jsonrpc", doc);

const char* name = doc["result"]["name"];
int major = doc["result"]["version"]["major"];
int minor = doc["result"]["version"]["minor"];
```

8.4.7 Summary

I hope you enjoyed this case study. I find it fascinating to see how easy it is to create a RESTful client with EthernetClient and ArduinoJson. Of course, you can do the same with WiFiClient or WiFiClientSecure.

If you look at the source files, you'll see that I created the class `KodiClient`, which provides another level of abstraction on top of `RestClient`. You could easily extend this class to add more features,such as controlling the playback.

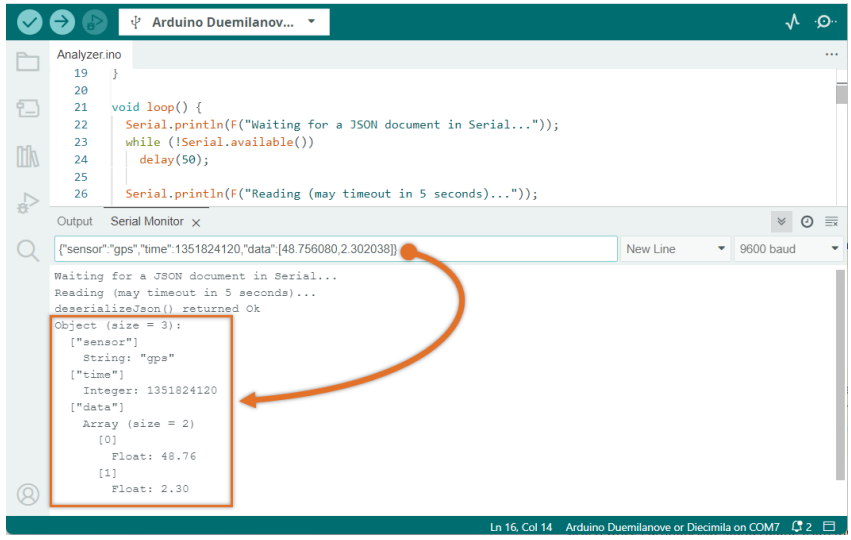
Another difference in the source files is that they include some error-checking that I omitted in the text. It also uses Flash string to save RAM.

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.



Given the following input:

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

The program prints the following output:

```
Object (size = 3):
  ["sensor"]
    String: "gps"
  ["time"]
    Integer: 1351824120
```

```
["data"]
  Array (size = 2)
    [0]
      Float: 48.76
    [1]
      Float: 2.30
```

The code must be recursive to print the content of arrays and objects inside other arrays and objects. Each recursion increases the indentation by two spaces so we can see the hierarchy.

The source code is in the `Analyzer` folder of the zip file.

8.5.2 Reading from the serial port

`Serial` is an instance of the class `HardwardSerial`, 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 `EmptyInput`. To avoid this timeout, we need to add a loop that waits for incoming characters:

```
// Loop until data is 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 the 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 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 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` statements:

```
// For clarity, I omitted the code for indenting the output

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<const char *>())
    dump(variant.as<const 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 a floating point type can always store an integral value. In other words, `is<long>()` implies `is<double>()`.

It may look like the `dump()` function is calling itself, but it's not the case. Indeed, this version of `dump()` takes a `JsonVariant` parameter but calls other versions of `dump()` with different parameter types. For example, if `variant` is a `bool`, `dump(JsonVariant)` calls `dump(bool)`. This is not recursion, just function overloading: the same name is used for different functions, and the compiler chooses the right one based on the type of the argument. I could have given different names to these functions, but it is a standard practice in C++ to use the same name for functions that do the same thing but with different types (think of `std::to_string()`, for example).

8.5.5 Printing values

Each overload of `dump()` is responsible for printing a value of a specific type.

Printing boolean, integral, and floating-point values is straightforward:

```
// For clarity, I omitted the code for indenting the output

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 through 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
  }
}
```

The last line calls `dump(JsonVariant)`; this is the recursion I teased you about earlier.

Arrays require a similar strategy:

```
void dump(const JsonArray &arr) {
  Serial.print("Array: ");

  int index = 0;
  // Iterate through 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 avoids 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.
- Flush the stream if `deserializeJson()` fails.
- 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; it's time to conclude this book.

Chapter 9
Conclusion

It's already the end of this book. I hope you enjoyed reading it and I hope you learn many things about Arduino, C++, and programming in general. Please let me know 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, Bupjae Lee, Carl Smith, Craig Feied, Daniel Travis, Darryl Jewiss, Dieter Grientschnig, Doug Petican, Douglas S. Basberg, Ewald Comhaire, Ezequiel Pavón, Gayrat Vlasov, Georges Auberger, Hendrik Putzek, HenkJan van der Pol James Fu, Jon Freivald, Joseph Chiu, Juergen Opschoref, Kent Keller, Konstantin Burkalev, Lee Bussy, Leonardo Bianchini, Louis Beaudoin, Matthias Waldorf, Matthias Wilde, Mike J Cheich, Morris Lewis, Nathan Burnett, Neil Lowden, Peter Dalmaris, Pierre Olivier Théberge, Robert Balderson, Ron VanEtten, Ted Timmons, 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 you rely on are continuously improved and won't be abandoned after a year or so.

Sincerely, Benoît Blanchon

Satisfaction survey

Please take a minute to answer a short survey.

Go to `arduinojson.survey.fm/book`

Index

-fno-exceptions 69
__FlashStringHelper 54
Adafruit IO 99, 113
AIO key 114
aJson 20
analogRead() 103
Arduino_JSON 19
ArduinoJson 160
ArduinoJson Assistant 197
ARDUINOJSON_DEFAULT_NESTING_LIMIT 226
ARDUINOJSON_ENABLE_COMMENTS 226
ARDUINOJSON_USE_LONG_LONG 175
ArduinoTrace 215
ArrayData 180
ATmega328 ..... 33, 37, 38, 56, 213
Attorney-Client 192
auto 68
AVR 209
Base64 13
BOM 225
BoundedReader<TInput> 195
BSON 13
BufferedStream 149
BufferingPrint 148, 230
canConvertFromJson() 157
CBOR 13
class 40
code samples 2
Codepoint 199
CollectionData 180
CollectionIterator 189
CollectionIterator* 180
Comments 13
Complex 161
containsKey() 73
Content-Length 108, 115
convertFromJson() 157
convertToJson() 157, 243
custom converters 156, 241
debugger 214
delete 45
DeserializationError 67, 219
DeserializationOption::Filter .. 129, 199, 253
DeserializationOption::NestingLimit 197
deserializeJson() . 67, 139, 152, 195, 253, 258, 274
deserializeMsgPack() 164, 195
design pattern 48, 79
DOM 20
Douglas Crockford 8
EmptyInput 219, 274
EscapeSequence 199
ESP32 217
ESP8266 .. 37, 56, 209, 213, 217, 239
ESP8266HTTPClient 84
EthernetClient 115
exceptions 69

F() 34, 55
FAT file system 84
fgetc() 154
File 83, 112, 245
finally 46
Flash 54, 59, 239
FloatParts 201
FloatTraits 199
fopen() 153
fputc() 153
fragmentation 36
fread() 154
free() 45, 210
fwrite() 153
git bisect 216
GitHub's API 65, 84
GraphQL 86
GSMClient 83
HardwareSerial 83, 111
Harvard architecture 31, 54
heap_caps_malloc() 143
heap_caps_realloc() 143
IncompleteInput 220
Infinity 10
InvalidInput 222
isNull() 78
isspace() 275
isUnbound() 188
jsmn 19
JSON streaming 139
JSON-RPC 262
json-streaming-parser 20
JsonArray::add<JsonObject>() 104
JsonArray::operator[] 105
JsonArray::remove() 105
JsonArray::size() 80
JsonDocument 66, 186
JsonDocument::add() 103
JsonDocument::as<T>() 245
JsonDocument::overflowed() 101
JsonDocument::set() 244
JsonDocument::to<JsonArray>() 105
JsonDocument::to<JsonObject>() 101
JsonDocumet::overflowed() 228
JSONLines 140
JsonObject::remove() 178
JsonPair 72
JsonString 72
JsonVariant 72, 187, 188
JsonVariant::as<T>() 84, 175, 234
JsonVariant::is<T>() 72, 81, 175
JsonVariantConst 188
Latch 196
LDJSON 140
LittleFS 239
LoggingPrint 146
LoggingStream 147
malloc() 45
MCVE 235
measureJson() 108, 200
measureJsonPretty() 108, 200
measureMsgPack() 200
memory leak. 46
MessagePack 13, 17, 162
MKR1000 247
NaN 10
NDJSON 140
new 45
NoMemory 226
NULL 43
null 120
null object 79
null pointer 43
nullptr 43
ObjectData 180
OpenWeatherMap 247
parseNumber() 199
Print 110, 151
Print::println() 234
PROGMEM 34, 54

RAII 46, 48, 245
ReadBufferingStream 148, 221
Reader<TInput> 195
ReadLoggingStream 146
Reddit's API 256
ResourceManager 186
SAX 20
serialized() 120
serializeJson() 106, 141, 200
serializeJsonPretty() 107, 200
serializeMsgPack() 163, 200
SlotId 180, 184
SoftwareSerial 83, 112
SPIFFS 239
SPIRAM, ...143
sprintf() 18, 59, 212
sprintf_P() 59
sscanf() 18
std::istream 83, 151
std::ostream 110, 151
std::shared_ptr 48
std::to_string() 277
std::unique_ptr 48
stdio.h 153
strcmp() 57, 209
strcpy() 54, 212
strcpy_P() 54
Stream 83, 151
Stream::available() 140
Stream::find() 135, 258
Stream::findUntil() 137, 258
Stream::peek() 275
Stream::println() 141
Stream::setTimeout() 222
StreamUtils 146, 148, 230
strftime() 156
String 109
string deduplication 118
string interning 54
StringAdapter 178
StringNode 177, 183
strlcpy() 70, 213, 242
strncpy() 213
strptime() 156
struct 40
TextFormatter 201
time.h 156
tm 156
TooDeep 226
tracing 215
TwoWire 83
Unicode 11
UTF-8 11
Utf8 199
Variable-length array 107
VariantData 183
VariantSlot 180, 183
Visitor pattern 190, 200
von Neumann architecture 31, 54
wandbox.org 17, 235
WiFiClient 83, 259
WiFiClientSecure 83, 259
Wire 112
Writer<TOutput> 200
XML 8