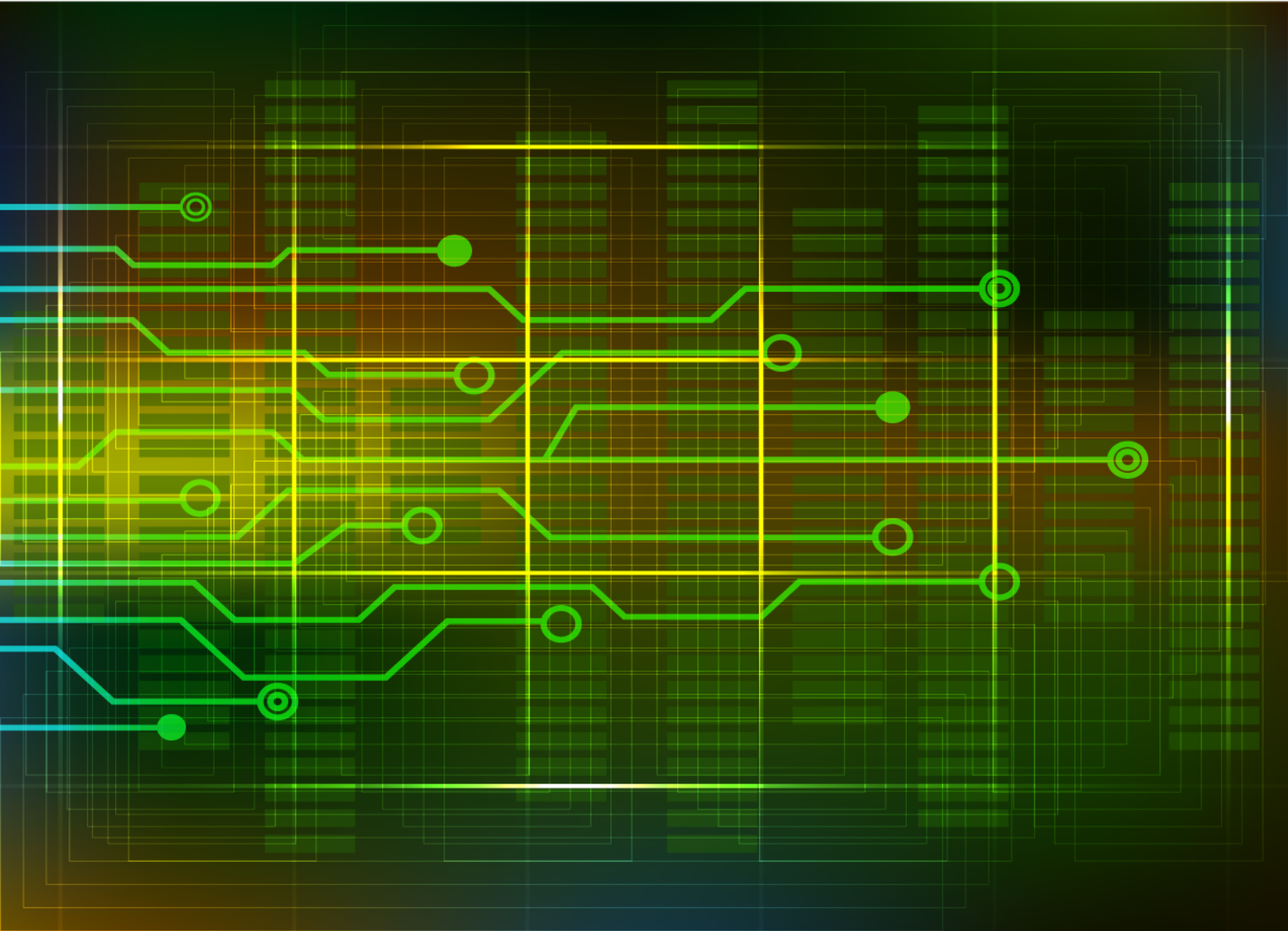


BENOIT BLANCHON
CREATOR OF ARDUINOJSON



Mastering ArduinoJson

Efficient JSON serialization for embedded C++



Mastering ArduinoJson

Efficient JSON serialization for embedded C++

Benoît Blanchon

This book is for sale at <http://leanpub.com/arduinojson>

This version was published on 2018-05-07



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2017 - 2018 Benoît Blanchon

This book is dedicated to the early users of ArduinoJson, who pulled me in the right direction. In particular, I want to thank the following GitHub users: @leventesen, @Slechtvalk, @shreyasbharath, @firepick1, and @sticilface.

Contents

Acknowledgments	i
1. Introduction	1
1.1 About this book	2
1.2 Introduction to JSON	3
1.3 Introduction to ArduinoJson	10
2. The missing C++ course	20
2.1 Why a C++ course?	21
2.2 Stack, heap, and globals	22
2.3 Pointers	26
2.4 Memory management	31
2.5 References	35
2.6 Strings	37
3. Deserialize with ArduinoJson	43
3.1 The example of this chapter	44
3.2 Parse a JSON object	45
3.3 Extract values from an object	49
3.4 Inspect an unknown object	52
3.5 Parse a JSON array	56
3.6 Extract values from an array	60
3.7 Inspect an unknown array	62
3.8 The zero-copy mode	65
3.9 Parse from read-only memory	68
3.10 Parse from stream	71
4. Serialize with ArduinoJson	77
4.1 The example of this chapter	78
4.2 Create an object	79
4.3 Create an array	82
4.4 Serialize to memory	86
4.5 Serialize to stream	89
4.6 Duplication of strings	94

CONTENTS

5. Inside ArduinoJson	97
5.1 Why JsonBuffer?	98
5.2 Inside StaticJsonBuffer	102
5.3 Inside DynamicJsonBuffer	106
5.4 Inside JsonArray	109
5.5 Inside JsonObject	112
5.6 Inside JsonVariant	116
5.7 Inside the parser	122
5.8 Inside the serializer	128
5.9 Miscellaneous	131
6. Troubleshooting	138
6.1 Program crashes	139
6.2 Deserialization issues	147
6.3 Serialization issues	152
6.4 Understand compiler errors	156
6.5 Common error messages	161
6.6 Log	165
6.7 Ask for help	169
7. Case Studies	171
7.1 Configuration in SPIFFS	172
7.2 OpenWeatherMap on mkr1000	180
7.3 Weather Underground on ESP8266	184
7.4 JSON-RPC with Kodi	189
7.5 Recursive analyzer	199
8. Conclusion	204

Acknowledgments

Adafruit is a trademark of Limor Fried and Adafruit Industries LLC. *Amazon Web Services* is a trademark of Amazon Technologies Inc. *Arduino* is a trademark of Arduino AG. *Atmel* is a trademark of Atmel Corporation. *Atollic* is a trademark of Atollic AB. *Azure* is a trademark of Microsoft Corporation. *Bitbucket* is a trademark of Atlassian Pty Ltd. *Dark Sky* is a trademark of The Dark Sky Company, LLC. *GitHub* is a trademark of GitHub Inc. *GitLab* is a trademark of GitLab B.V. *Google Cloud* is a trademark of Google LLC. *IAR Embedded Workbench* is a trademark of I.A.R. Systems AB. *IFTTT* is a trademark of IFTTT Inc. *ImperiHome* is a trademark of Everytygo SAS. *Jeedom* is a trademark of David Bonnamour. *Keil* is a trademark of ARM Inc. *Kodi* is a trademark of the XBMC Foundation. *MPLAB* is a trademark of Microchip Technology Inc. *openHAB* is a trademark of Kai Kreuzer. *OpenWeatherMap* is a trademark of OpenWeatherMap, Inc. *SD* is a trademark of SD-3C, LLC. *Temboo* is a trademark of Temboo Inc. *ThingSpeak* is a trademark of The MathWorks Inc. *Weather Underground* is a trademark of The Weather Company LLC. *Xively* is a trademark of LogMeIn Inc. *Yahoo!* is a trademark of Yahoo! Inc. and Oath Inc. *Zapier* is a trademark of Zapier Inc.

[@bcendet](#) designed the ArduinoJson logo.

Iulia Ghimisli designed the cover of this book.

1. Introduction

“

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

– Douglas Crockford, JavaScript: The Good Parts

1.1 About this book

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

Let's see how this book is organized:

1. The first chapter is an introduction to JSON and ArduinoJson.
2. The second chapter is a quick C++ course. It 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. The third chapter is a step-by-step tutorial that teaches how to use ArduinoJson to deserialize a JSON document.
4. The fourth chapter is another tutorial, but for serialization.
5. The fifth chapter opens the hood and shows how ArduinoJson works.
6. The sixth chapter is a troubleshooting guide. If you don't know why your program crashes or why compilation fails, this chapter is for you.
7. The last chapter takes several concrete project examples and explains how they work. It shows the best coding practices in various situations.

This version of the book covers ArduinoJson 5.13.

You can download the code samples from arduinojson.org/book/sketchbook.zip

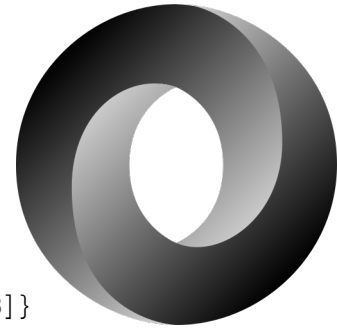
1.2 Introduction to JSON

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:

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



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

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

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

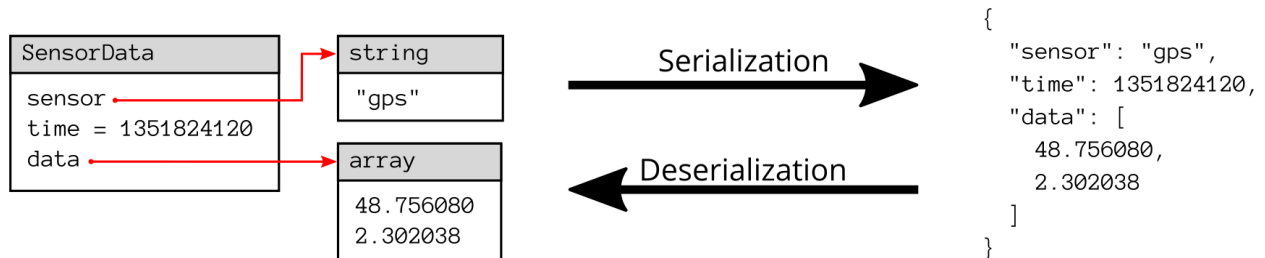
```
{
  "sensor": "gps",
  "time": 1351824120,
  "data": [
    48.756080,
    2.302038
  ]
}
```

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

What is serialization?

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



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 a program.

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

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

- Weather forecast
 - [Dark Sky](#), formerly known as forecast.io
 - [OpenWeatherMap](#) (we'll see an example [in case studies](#))
 - [Weather Underground](#) (we'll see an example [in case studies](#))
 - [Yahoo!](#) (we'll see an example [in the third chapter](#))
- Internet of Thing (IoT)
 - [Adafruit IO](#) (we'll see an example [in the forth chapter](#))
 - [ThingSpeak](#)
 - [Temboo](#)
 - [Xively](#)
- Cloud providers
 - [Amazon Web Services](#)
 - [Google Cloud Platform](#)

- [Microsoft Azure](#)
- Code hosting
 - [Bitbucket](#)
 - [GitHub](#)
 - [GitLab](#)
- Home automation
 - [Jeedom](#)
 - [openHAB](#)
 - [ImperiHome](#)
- Task automation
 - [IFTTT](#)
 - [Zapier](#)

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,” or “REST API.”

History of JSON

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

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

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

As curious as it sounds, the JSON notation was “discovered” as a hidden gem in the JavaScript language itself. This discovery is attributed to Douglas Crockford and became popular in 2008 in 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 known as *The Fat-Free Alternative to XML*.



Why is JSON so popular?

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

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

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

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

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

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

The JSON syntax

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

JSON documents are composed of the following values:

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

Booleans

A *boolean* is a value that can be either `true` or `false`. It must not be surrounded by quotes; 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, many projects, including ArduinoJson, support them.

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, 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.

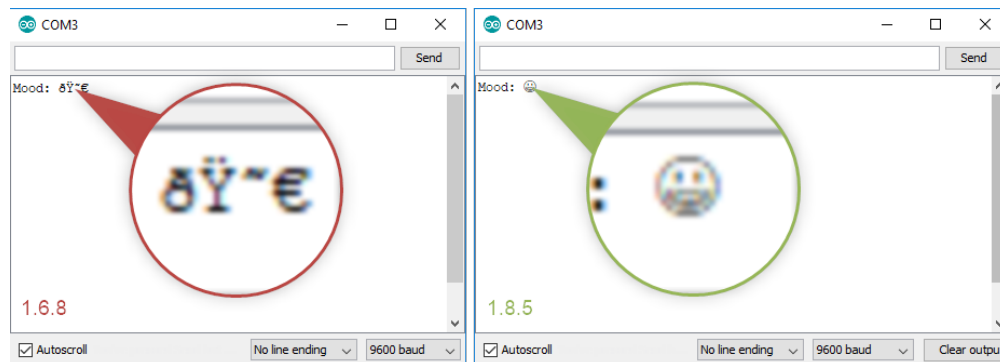
JSON has a notation to specify an extended character using a UTF-16 escape sequence, for example "\uD83D". However, very few projects use this syntax; many JSON parsers, including ArduinoJson,

don't support it. Instead, most projects use UTF-8 in their JSON documents, so translating to UTF-16 escape sequence is not necessary.



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.



UTF-8 in different versions of Serial Monitor

Arrays

An *array* is a sequence of values.

Example:

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

Syntax:

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

Objects

Objects are just 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 for its key
- Keys are surrounded by double quotes (")

In a single object, each key should be unique; you're not allowed to have several key-values pairs with the same key.



JSON vs JavaScript

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

Misc

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

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

Binary data in JSON

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

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

As an example, the title of the book encoded in base64 is `TWFzdGVyaW5nIEFyZHVpbm9Kc29u`.



Binary JSON?

Several alternative data formats claim to be the "binary version of JSON," the two most famous are BSON and MessagePack. All these formats solve the problem of storing binary data in JSON document.

1.3 Introduction to ArduinoJson

What is ArduinoJson?

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

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

ArduinoJson can be used outside of Arduino, as soon as you have a C++ compiler. Here are some alternative platforms where you can use ArduinoJson:

- [Atmel Studio](#)
- [Atollic TrueSTUDIO](#)
- [Energia](#)
- [IAR Embedded Workbench](#)
- [MPLAB](#)
- [PlatformIO](#)
- [Keil µVision](#)

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

What makes ArduinoJson different?

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

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



The main strength of ArduinoJson is its memory management strategy. In order to avoid fragmentation and overhead caused by dynamic memory allocations, ArduinoJson allocates only one buffer and then fill this memory. You'll learn more on this topic in the chapter [Inside ArduinoJson](#).

This fixed-allocation strategy makes it suitable for real-time applications where execution time must be predictable. Indeed, if you use a `StaticJsonBuffer`, the whole JSON serialization and deserialization is done in bounded time.

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

ArduinoJson is self-contained. It doesn't depend on any library. In particular, it doesn't depend on Arduino, that's why you can use it in any C++ project. This feature allows to compile and run your unit test 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.

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) when compiled for a computer. The following table shows how types match:

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

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

Does size really matters?

Let's take a concrete example to show how important are program size and memory usage. Suppose we have an Arduino Duemilanove, 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 12,234 bytes (39%) of program storage space. Maximum is 30,720 bytes. Global variables use 742 bytes (36%) of dynamic memory, leaving 1,306 bytes for local variables. Maximum is 2,048 bytes.

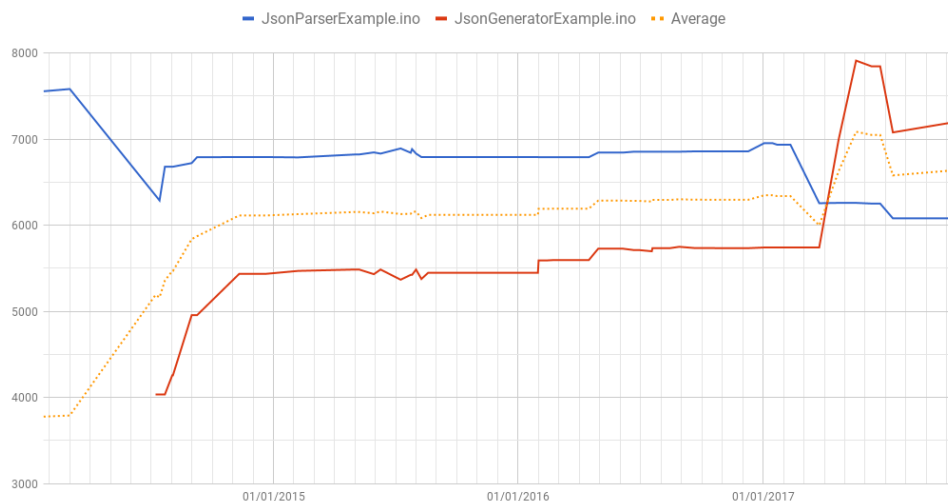
Yep. That is right. 39% of program space and 36% of memory is already taken by the skeleton. From that baseline, each new line of code increases these numbers until you need to get a bigger microcontroller.

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

Sketch uses 14,638 bytes (47%) of program storage space. Maximum is 30,720 bytes. Global variables use 786 bytes (38%) of dynamic memory, leaving 1,262 bytes for local variables. Maximum is 2,048 bytes.

ArduinoJson added 2404 bytes Flash and 44 bytes of RAM to the program, which is very small when you consider all the features that it supports. The library represents only 8% of the capacity but enables a wide range of applications.

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



Evolution of code size

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

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 `scanf()`. However, as soon as there are nested objects and arrays with variable size, you need a library. Here are three alternatives for Arduino.

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 JSON document, then it generates a list of tokens (object, array, string...), each token having a start and end positions which are indexes in the input string.

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

aJson

[aJson](#) is a full-featured JSON library for Arduino.

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

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

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

json-streaming-parser

[json-streaming-parser](#) is a library for parsing potentially huge JSON streams on devices with scarce memory.

It only supports deserialization but can read a JSON input that is bigger than the RAM of the device. ArduinoJson cannot do that, but we'll see a workaround in [the case studies](#).

json-streaming-parser is very different from ArduinoJson. Instead of deserializing the JSON document into a data structures, 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.

I often recommend this library when ArduinoJson is not suitable.

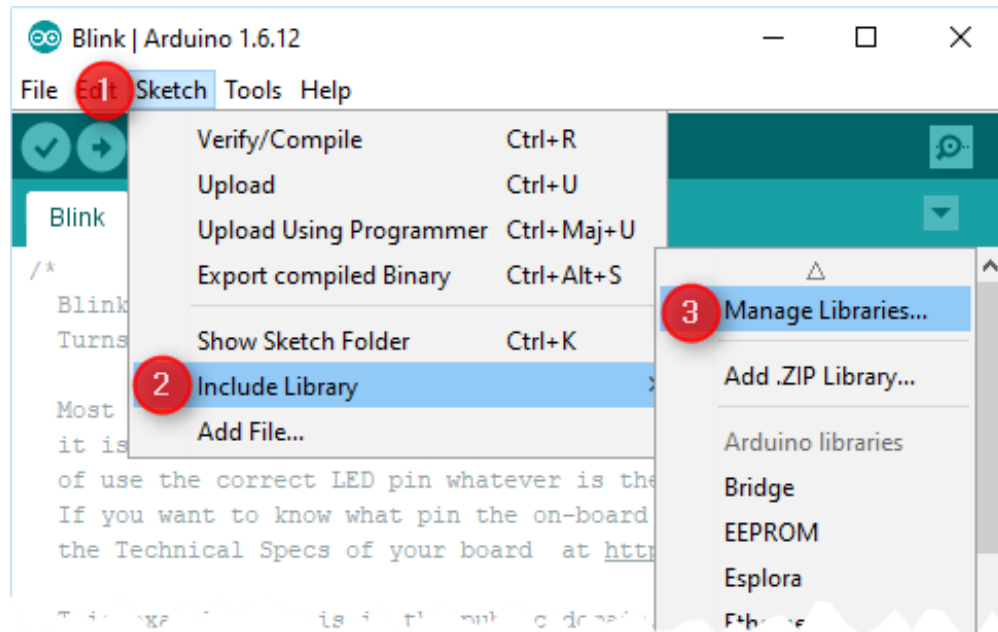
How to install ArduinoJson

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

Using the Arduino Library Manager

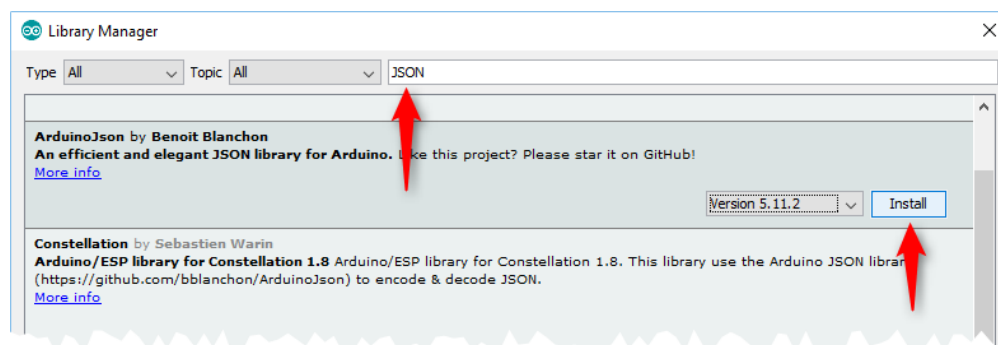
If you use the Arduino IDE version 1.6 or newer, you can install ArduinoJson directly from the IDE, thanks to the “Library Manager.” The Arduino Library Manager lists all installed libraries; it allows to install new ones and to update the ones that are already installed.

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



Arduino IDE, Manage Libraries

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

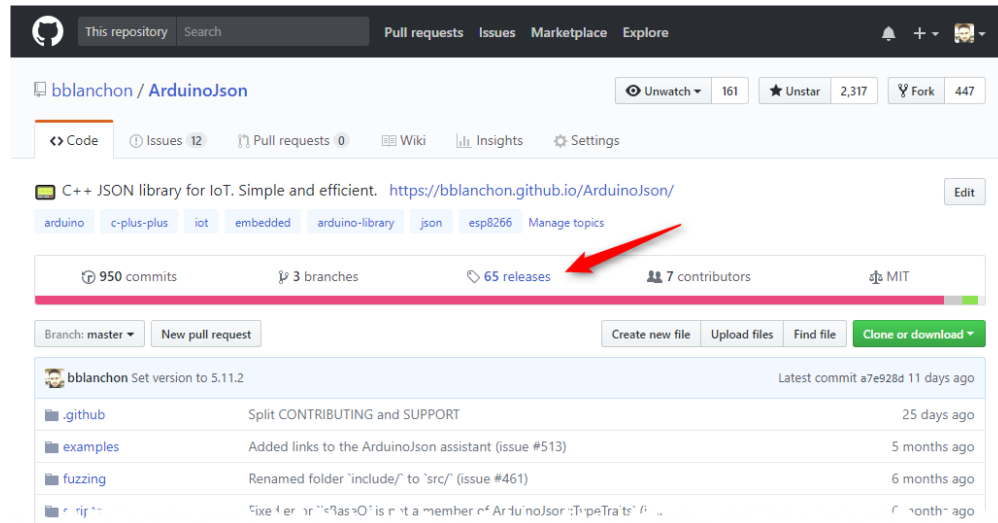


Arduino Library Manager

Using the “single header” distribution

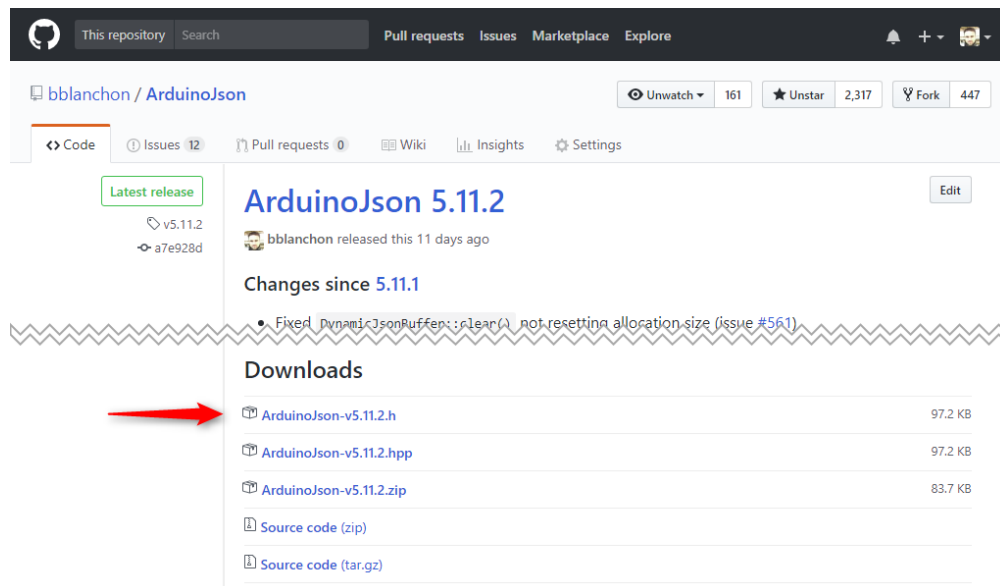
If you don’t use the Arduino IDE, the simplest way to install ArduinoJson is to put the entire source code of the library in your project folder. Don’t worry! It’s just one file :-)

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



ArduinoJson on GitHub

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



Download section in Release page

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

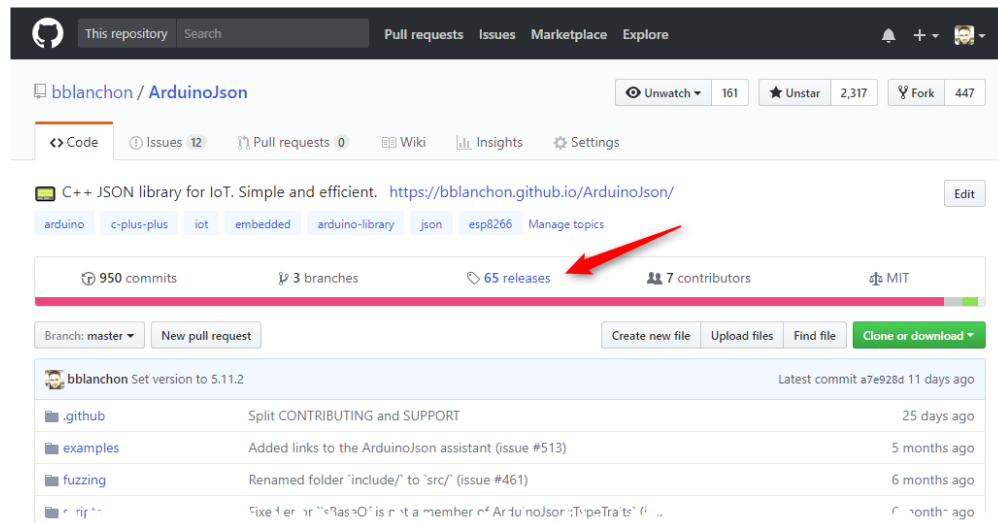
Save this file in your project folder, alongside with your source code.

As you can see, there is also a .hpp file that is identical to the .h file, except that everything is kept inside the ArduinoJson namespace.

Using the Zip file

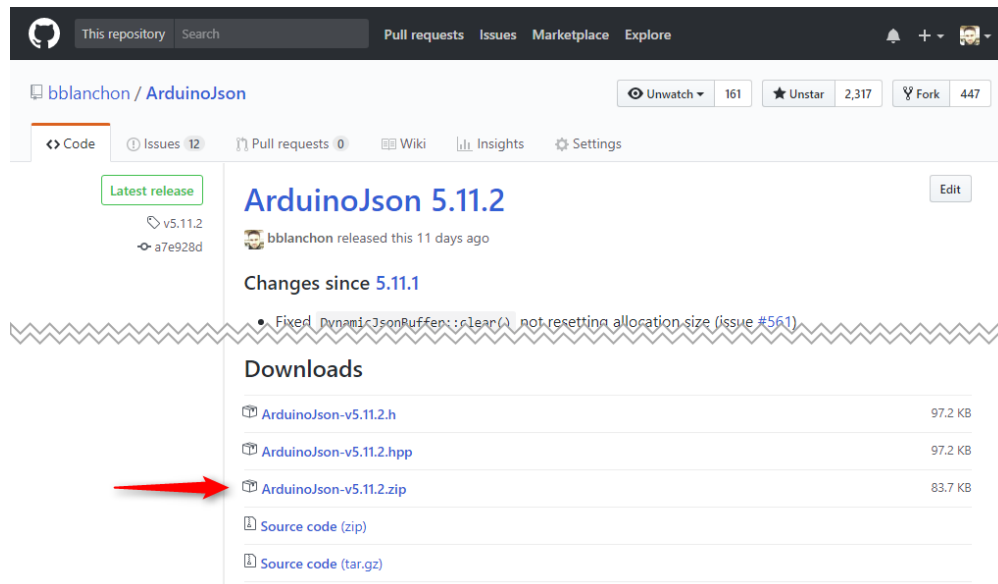
If you use an old version of the Arduino IDE, you may not be able to use the Library Manager. In this case, you need to do the job of the Library Manager yourself by downloading and unpacking the library into the right folder.

Go to the [ArduinoJson GitHub page](https://github.com/bblanchon/ArduinoJson), then click on “Releases.”



ArduinoJson on GitHub

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



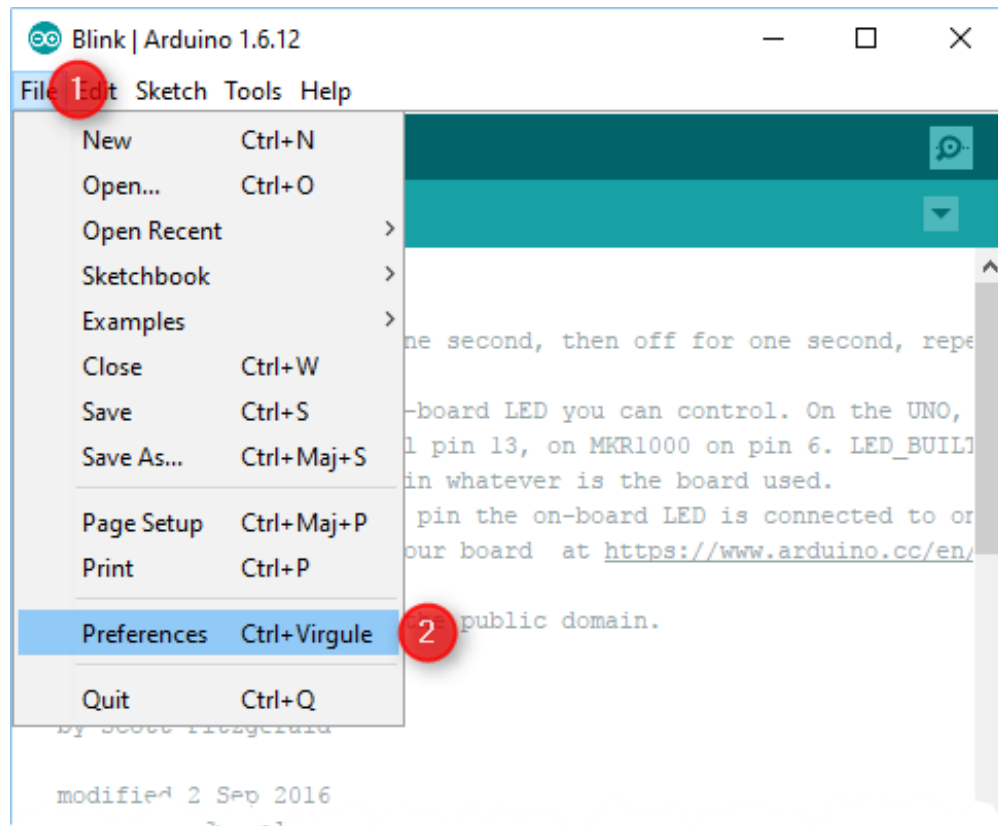
Download section in Release page

Click on the ArduinoJson-vX.X.X.zip file to download the package for Arduino. Don't use the link

“Source code (zip)” as it includes unit tests and a few other things you don’t need to use the library. Using your favorite file archiver, unpack the zip file into the Arduino’s libraries folder, which is:

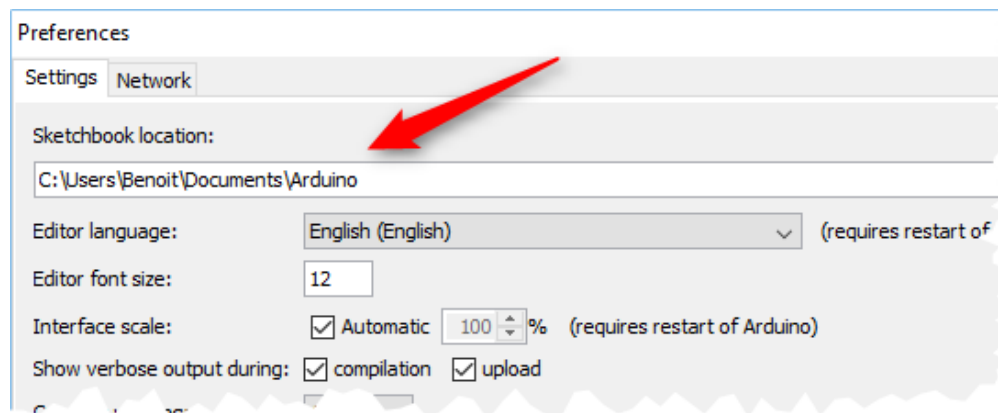
<Arduino Sketchbook folder>/libraries/ArduinoJson

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



Arduino preferences in the menu

The setting is named “Sketchbook location.”

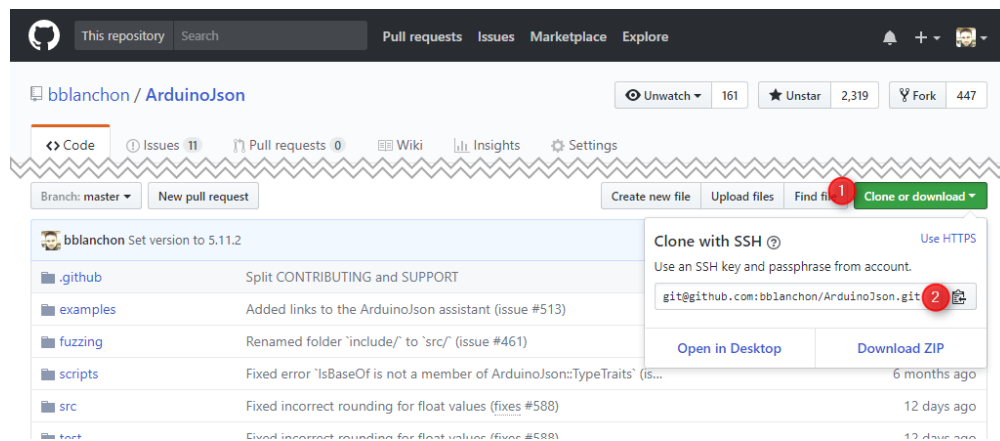


Arduino Sketchbook configuration

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.”

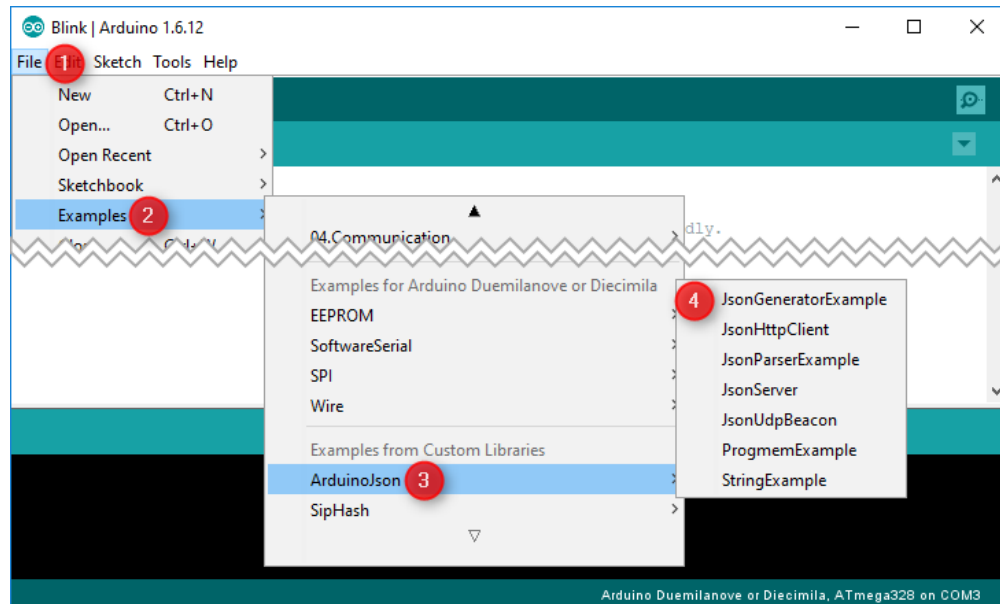


GitHub's button “Clone or download”

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

The examples

If you use the Arduino IDE, there is a quick access to the examples from the “File” / “Examples” menu.



Arduino Example menu

If you don't use Arduino IDE, or if you installed ArduinoJson as a single-header, you can [see the examples online](#).

There are seven examples provided with ArduinoJson:

1. `JsonGeneratorExample` shows how to encode JSON and write the result to the serial port.
2. `JsonParseExample` shows how to parse JSON and print the result to the Serial port.
3. `JsonHttpClient` shows how to perform an HTTP request and parse the JSON response.
4. `JsonServer` shows how to implement an HTTP server that returns the status of analog and digital inputs in a JSON response.
5. `JsonUdpBeacon` shows how to send UDP packets with a JSON payload.
6. `ProgmemExample` show how to use Flash strings with ArduinoJson.
7. `StringExample` shows how to use the `String` class with ArduinoJson.
8. `ConfigFile` shows how to save a JSON document to an SD card.

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 a lack of understanding of some of the C++ fundamentals. That's why, before looking at ArduinoJson in detail, we're going to learn the elements of C++ that are necessary to understand the remaining of the book.

There are a lot of beginner's guides to Arduino, they introduce you to the C++ syntax, but omit to explain how it works behind the scene. 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 memory is managed?
- What's a pointer?
- What's a reference?
- How are string implemented?

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

In this book, I assume that you already know:

1. How to program in another object-oriented language like Java or C#, and the following concepts:
 - class and instance
 - constructor and destructor
 - scope: global, function or block
2. How to do basic stuffs with Arduino:
 - compile and upload
 - Serial Monitor
 - `setup()` / `loop()`
3. How to write simple C / C++ programs:
 - `#include`
 - `class` / `struct`
 - `void`, `int`, `float`, `String`
 - functions



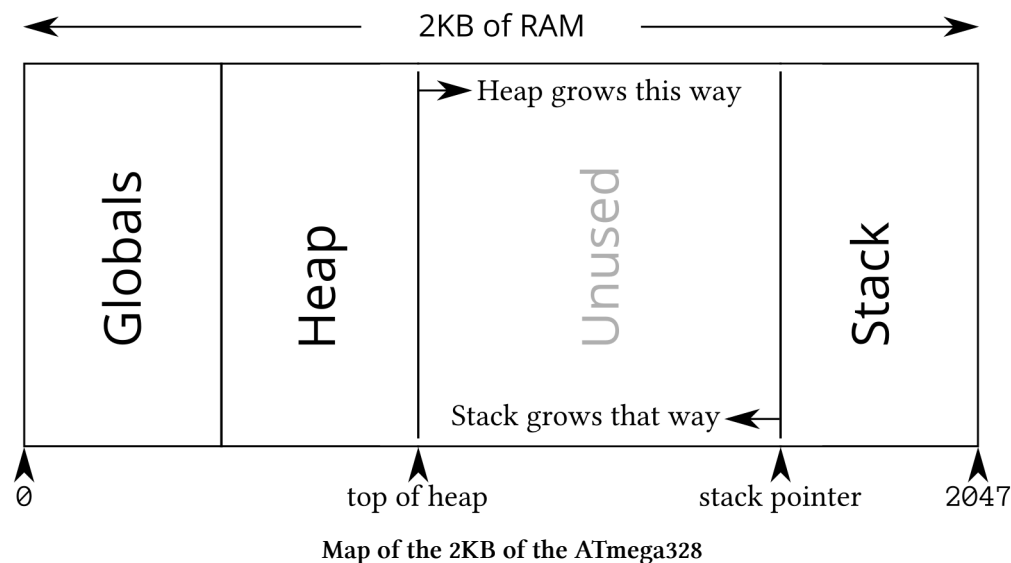
2.2 Stack, heap, and globals

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

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

To see how 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 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 standard libraries) cuts this huge array into three areas that they use for different kinds of data.



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

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.

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

Here is a program that declares a global variable:

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

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

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

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

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

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

Heap

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

To create a variable in the heap, a program needs to allocate it explicitly. If you're used to C# or Java, it is the same as variables instantiated via a call to `new`. In C++, it is the job of the program to release the memory. Unlike C# or Java, there is no garbage collector to manage the heap. Instead, the program must call a function to release the memory.

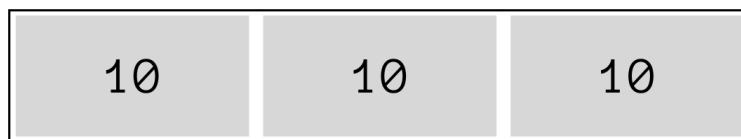
Here is a program that performs allocation and deallocation in the heap:

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

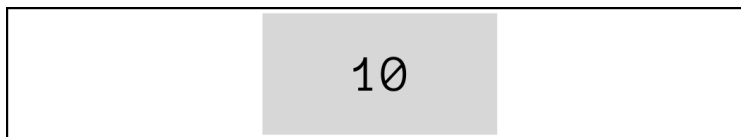
I insist on the fact that it is the role of the *program* and not the role of the *programmer* to manage the heap. Indeed, it is a common misunderstanding that, in C++, memory must be managed manually by the programmer; in reality, the language does that for us, but we'll see that in the [section dedicated 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 is 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 is called “heap fragmentation,” and it is the bane of embedded systems because it wastes the precious RAM.



Heap is optional

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



When heap is forbidden

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

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

Stack

The “stack” contains:

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

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

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

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

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

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



Stack size

While it is not the case for the ATmega328, many architectures have a limit on the size of the stack. For example, it is limited to 4KB on the ESP8266, although it can be adjusted by changing the configuration of the compiler.

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



Code is good but crashes anyway?

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

2.3 Pointers

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

What is a pointer?

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

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

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

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 set a pointer to 0x100 and use 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`, a star (*) is used to declare a pointer. At the left of the star, we need to specify the type of value pointed by the pointer; it's the type of value can we read from that pointer.

Then, we see that the star is also used to read the value pointed by the pointer, it is called "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 to an integer:

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

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

Pointers and arrays

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

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

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

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

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

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

Taking the address of a variable

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

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

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

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

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

Pointer to class and struct

In C++, object classes are declared with `class` or `struct`. The two keywords only differ by the default accessibility of members: for `class`, it's private; for `struct` is public.



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, it is allocated in the (managed) heap; a `struct` is a value type, 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 which decides if the variable goes in the heap or the stack.

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

Here is a program that uses both operators:

```
// Declare a structure
struct MyStruct {
    int value;
};

// Create an instance in the stack
MyStruct s;

// Set the member directly
s.value = 1;

// Get a pointer to the instance
MyStruct* p = &s;

// Modify the member via the pointer
p->value = 2;
```

The null pointer

Zero is a special value to mean an empty pointer, just as you'd use `null` in Java or C#.

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 clear 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 :-(
}
```

Why using pointers?

In C, pointers are used for the following tasks:

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

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

2.4 Memory management

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

`malloc()` and `free()`

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

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

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

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

`new` and `delete`

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

Here is a program that uses these operators:

```
// Declare a class  
struct MyStruct {  
    MyStruct() {} // constructor  
    ~MyStruct() {} // destructor  
    void myFunction() {} // member function  
};  
  
// Instantiate the class in the heap  
MyStruct* str = new MyStruct();  
  
// Call a member function  
str->myFunction();  
  
// Destruct the instance  
delete str;
```

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

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



No finally in C++

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

This is a conscious design decision from the author of C++. He wants to encourage programmers to use a superior technique called RIAA; more on that later.

Smart pointers

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

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

Here is an example:

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

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

```

    }

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

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

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

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

// calls 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 to overload built-in operators, like `->`, so that the smart pointer keeps the same semantics as the raw pointer.

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



unique_ptr and shared_ptr

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

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

RAII

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

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



The String class

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



If you must remember only one thing from this book

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

2.5 References

What is a reference?

A “reference” in C++ 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:

```
// Create an integer
int v = 42;

// Create a reference to v
int& r = v;

// Modify v via the reference
r = 666;

// Now v == 666
```

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

Differences with pointers

The example above could be rewritten with a pointer:

```
// Create an integer
int v = 42;

// Create a pointer pointing to v
int* r = &v;

// Modify v via the pointer
*r = 666;

// Now v == 666
```

Pointer and references are very similar, except that references keep the value semantics. 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 the code is compiled, there are the same thing; they translate to the same assembler code.

Rules of references

References provide additional compile-time safety compared to pointers:

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 a warning when you create a reference to a temporary because it is too risky.

Common problems

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

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

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

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

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

Usage for references

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

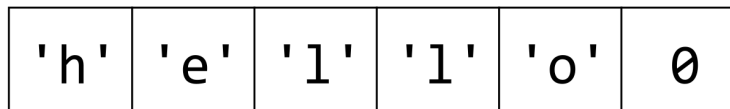
1. To pass parameters to function when copying is expensive.
2. To keep a dependency on an object that we do not own.

2.6 Strings

How string are 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 character terminated by a zero. The zero marks the end of the string and is called a null-terminator.

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



The string "hello" in memory

As you see, a string of 5 characters is represented with an array of 6 bytes.



Outside of Arduino

We saw the most common way to encode strings in C++, but 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.

String literals in RAM

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

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

We'll see how these forms differ next, but before that, let's focus on the common part: the string literal "hello".

The three expressions above cause the same 6 bytes to be added to the "globals" area of the RAM. As we saw, this area should be as small as possible because it limits the remaining RAM from the rest of the program.

String literals in Flash

To reduce the size of the "globals" sections, we can instruct the compiler to keep the strings within the Flash memory, alongside with the program.

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

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

As you see, I made two changes:

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

The string literal “hello” is now stored in the Flash memory, the pointer `s` is not a regular pointer because its value is an address in the program space, not in the memory space.

To access this string, you need to use special functions (like `strcpy_P()`), or rely on classes that do that for you (like `String` and `Serial`). Reading program memory causes a significant overhead in code size and speed. I recommend using this technique for long strings that are rarely used, like logs.



The `F()` macro

There is a short-hand syntax to declare literal in Flash memory without using the `PGMEM` attribute: the `F()` macro. This macro is very handy because you can use it in-place, like this:

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



`__FlashStringHelper`

Whether it’s in `String`, in `Serial`, or in `ArduinoJson`, every function that takes a string as a parameter needs a way to know if the pointer is a location in the RAM or in the Flash. To differentiate between the two address spaces, a special pointer type is used: `const __FlashStringHelper*`. If you declared a `char[] PROGMEM`, like above, you need to cast explicitly:

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

The cast is not required if you use the `F()` macro, as it does the cast for you.



Use with care!

You can quickly shoot yourself in the foot with Flash strings. Many functions (including `ArduinoJson` and `String`) need to copy the entire string to RAM before using it. You need to make sure that only a few copies are in RAM at a given time otherwise you’d end up using more RAM than with regular strings.

Pointer to the “globals” section

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

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

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



Segmentation fault

If you remove the `const` keyword, the compiler emits a warning, because you are not supposed to modify the content of a string literal. Whether it is possible or not to modify the string depends on the platform. On an ATmega328, it will work, because there is no memory protection. But on other platforms, such as a PC or an ESP8266, the program will cause an exception.

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.

A copy in the stack

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

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

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



Code smell

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



Prefer uninitialized arrays

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

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

A copy in the heap

We just saw how to get a copy of a string in the stack, now let's see how to copy in the heap.

The third syntax presented was:

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

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

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

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



Code smell

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

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



From Flash to heap

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

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

This is a good usage of the `String` class; however don't abuse it. Remember that there is dynamic allocation and duplication behind the scenes.

A word about the `String` class

I almost never use the `String` class in my programs.

Indeed, `String` relies on the two things I try to avoid in embedded code:

1. Dynamic memory allocation
2. Duplication

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

Here is an example:

```
int answer = 42;

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

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

Too bad a `sprintf_P()` (which would take a Flash string as the second parameter) doesn't exist.



Everything you know is wrong

If you come from Java or C#, using a `String` class is the more natural approach. Part of the process of moving from a high-level application developer to embedded developer is to change your habits.

Yes, fixed-size strings are better than variable-size strings. There are faster, smaller and more reliable.

That's the end of our C++ course. There are still a lot to cover, but it's not the goal of this book. What's important now is that you understand how memory is managed so that you don't fall in the usual pitfalls. By the way, I may write a "C++ course for Arduino" in the future; please fill the [survey](#) at the end of the book if you're interested.

3. Deserialize with ArduinoJson



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

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

3.1 The example of this chapter

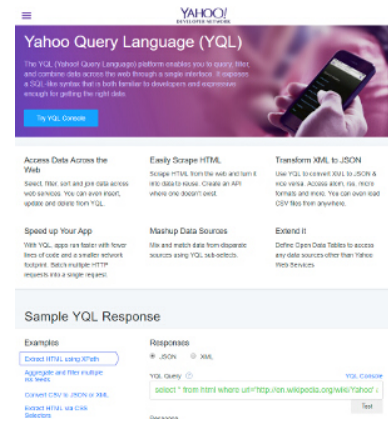
Now that your familiar with JSON and C++, we're going learn how to use ArduinoJson. This chapter explains everything there is to know about deserialization. As we've seen, deserialization is the process of converting a sequence of byte into a memory representation. In our case, it means converting a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

In this chapter, we'll use a JSON response from Yahoo Query Language (YQL) as an example. YQL is a web service that allows fetching data from the web in a SQL-like syntax. It is very versatile and can even retrieve data outside of the Yahoo realm. Here are some examples of what you can do with YQL:

- download weather forecast (we'll do that in this chapter)
- download market data
- scrap web pages via XPath or CSS selectors
- read RSS feeds
- search the web
- search on a map

For most applications, you don't need to create an account.

For our example, we'll use a 3-day weather forecast of the city of New York. We'll begin with a simple program, and add complexity one bit at a time.



The YQL front page

3.2 Parse a JSON object

Let's begin with the most simple situation: a JSON document in memory. More precisely, our JSON document resides in the stack in a writable location. This fact is going to matter, [as we will see later](#).

The JSON document

Our example is today's weather forecast for the city of New York:

```
{  
  "date": "08 Nov 2017",  
  "high": "48",  
  "low": "39",  
  "text": "Rain"  
}
```

As you see, it's a flat JSON document, meaning that there is no nested object or array.

It contains the following piece of information:

1. date is the date for the forecast: November 8th, 2017
2. high is the highest temperature of the day: 48°F
3. low is the lowest temperature of the day: 39°F
4. text is the textual description of the weather condition: "Rain"

There is something quite unusual with this JSON document: the integer values are actually strings. Indeed, if you look at the high and low values, you can see that they are wrapped in quotes, making them strings instead of integers. Don't worry, it is a widespread problem, and ArduinoJson handles it appropriately.

Place the JSON document in memory

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

```
char input[] = "{\"date\":\"08 Nov 2017\", \"high\":\"48\", \"low\":\"39\", \"  
               \"text\":\"Rain\"}";
```

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

Introducing `JsonBuffer`

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

Here is how it work:

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

The memory of the `JsonBuffer` can be either in the stack or in the heap, depending on the derived class you choose. If you use a `StaticJsonBuffer`, it will be in the stack; if you use a `DynamicJsonBuffer`, it will be in the heap.

A `JsonBuffer` is responsible for reserving and releasing the memory used by `ArduinoJson`. It is an instance of the RAII idiom that [we saw in the previous chapter](#).



StaticJsonBuffer in the heap

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

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

How to specify the capacity of the buffer?

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

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

```
DynamicJsonBuffer jb(capacity);
```

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

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

```
StaticJsonBuffer<capacity> jb;
```

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

How to determine the capacity of the buffer?

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

To answer this question, you need to know what ArduinoJson will store in the `JsonBuffer`. ArduinoJson needs to store a tree of data structures that mirrors the hierarchy of objects in the JSON document. In other words, it contains objects describing the JSON document and these objects are linked one to another, in the same way as they are in the JSON document.

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

ArduinoJson provides macros for computing precisely the capacity of the JSON buffer. Here is how to compute the capacity of our JSON document composed of only one object containing four elements:

```
// Enough space for one object with four elements  
const int capacity = JSON_OBJECT_SIZE(4);
```

On an ATmega328, an 8-bit processor, this expression evaluates to 44 bytes. The result would be significantly bigger on a 32-bit processor; for example, it would be 72 bytes on an ESP8266.

StaticJsonBuffer OR DynamicJsonBuffer?

For our example, running on an Arduino Ethernet, I'm going to use a `StaticJsonObject` for the following reasons:

1. The buffer is tiny (44 bytes).
2. The RAM is very scarce on an ATmega328 (only 2KB).
3. The size of the stack is not limited on an ATmega328.

Here is our program so far:

```
const int capacity = JSON_OBJECT_SIZE(4);  
StaticJsonBuffer<capacity> jb;
```



Don't forget `const`!

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

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

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

Parse the object

Now that the `JsonBuffer` is ready, we can parse the input. To parse an object, we just need to call `JsonBuffer::parseObject()`:

```
JsonObject& obj = jb.parseObject(input);
```

And now we're ready to extract the content of the object!



JsonBuffer returns references

As you see, it's not a `JsonObject` that is returned by `parseObject()` but a reference to a `JsonObject`. Indeed, the `JsonObject` resides inside the `JsonBuffer`, and we don't want to make a copy.

Verify that parsing succeeds

The first thing we can do is to verify that `JsonBuffer::parseObject()` actually succeeded. To do that, we just need to check the return value of `JsonObject::success()`:

```
if (obj.success()) {  
    // parseObject() succeeded  
} else {  
    // parseObject() failed  
}
```

ArduinoJson is not very verbose when parsing fails: the only clue is this boolean. This design was chosen to make the code small and prevent users from bloating their code with error checking. If I were to make that decision today, the outcome would probably be different.

However, there are a limited number of reasons why parsing could fail. Here are the three most common causes, by order of likelihood:

1. The input is not a valid JSON document.
2. The `JsonBuffer` is too small.
3. There is not enough free memory.



More on arduinojson.org

For an exhaustive list of reasons why parsing could fail, please refer to this question in the FAQ: [“Why parsing fails?”](#)

3.3 Extract values from an object

In the previous section, we used `ArduinoJson` to parse a JSON document. We now have an in-memory representation of the JSON object, and we can inspect it.

Extract values

There are multiple ways to extract the values from a `JsonObject`; we'll see all of them.

Here is the first and simplest syntax:

```
const char* date = obj["date"];
int         high = obj["high"];
int         low  = obj["low"];
const char* text = obj["text"];
```

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.

Explicit casts

Not everyone likes implicit casts, mainly because it messes with parameter type deduction and with the `auto` keyword.



The `auto` keyword

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

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

```
auto date = obj["date"].as<char*>();
auto high = obj["high"].as<int>();
auto low  = obj["low"].as<int>();
auto text = obj["text"].as<char*>();
```



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

We could have used `as<const char*>()` instead of `as<char*>()`, it's just shorter that way. The two functions are identical: in both cases, the returned type is `const char*`.

Using `get<T>()`

As operator overloading is also a matter of taste, ArduinoJson offers a third syntax using a method instead of the subscript operator.

Here is again the same code, with this syntax:

```
auto date = obj.get<char*>("date");
auto high = obj.get<int>("high");
auto low  = obj.get<int>("low");
auto text = obj.get<char*>("text");
```



Which syntax to use?

We saw three different syntaxes to do the same thing. They are all equivalent and lead to the same executable. None is better than the other; it's just a matter of coding style. You can choose whichever you are comfortable with.

When values are missing

We saw how to extract values from an object, but we didn't do error checking; now let's talk about what happens when a value is missing.

When that happens, ArduinoJson returns a default value, which depends on the type:

Type	Default value
const char*	nullptr
float, double	0.0
int, long...	0
String	""
JsonArray	JsonArray::invalid()
JsonObject	JsonObject::invalid()

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



No exceptions

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

Change the default value

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

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

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

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

`strcpy()`, 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)  
    strcpy(hostname, configHostname, 32);  
else  
    strcpy(hostname, "arduinojson.org");
```

This syntax is new in ArduinoJson 5.12, and it’s only available when you use the subscript syntax (`[]`). We’ll see a complete example in the [case studies](#).

3.4 Inspect an unknown object

So far, we extracted values from an object that we know in advance. Indeed, we knew that the JSON object had four members (`date`, `low`, `high` and `text`) and they were all strings. In this section, we'll see what tools are at our disposition when dealing with unknown objects.

Enumerate the keys

The first thing we can do is look at all the keys and their associated values. In ArduinoJson, a key-to-value association, or a key-value pair, is represented by the type `JsonPair`.

We can enumerate all pairs with a simple for loop:

```
// Loop through all the key-value pairs in obj
for (JsonPair& p : obj) {
    p.key // is a const char* pointing to the key
    p.value // is a JsonVariant
}
```

Three comments on this code:

1. I explicitly used a `JsonPair` to emphasize the type, but you can use `auto`.
2. I used a reference `&` to prevent a (small) copy and to be able to modify the value if I need.
3. The value associated with the key is a `JsonVariant`, a type that can represent any JSON type.



When C++11 is not available

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

```
for (JsonObject::iterator it=obj.begin(); it!=obj.end(); ++it) {
    it->key // is a const char* pointing to the key
    it->value // is a JsonVariant
}
```

Detect the type of a value

As we saw, `ArduinoJson` stores values in a `JsonVariant`. This class can hold any JSON value: string, integer... A `JsonVariant` is returned when you call the subscript operator, like `obj["text"]` (this statement is not 100% accurate, but it's conceptually a `JsonVariant` that is returned).

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

For example, if we want to test that the value in our object is a string:

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

If you use this with our JSON document from Yahoo Weather, you will find that all values are strings. Indeed, as we said earlier, there is something special about this example: integers are wrapped in quotes, making them strings. If you remove the quotes around the integers, you will see that the corresponding `JsonVariants` now contain integers instead of strings.



Alternative syntax for `is`

If you're testing the type of a value whose key is known, you can use either of the two following syntax:

```
obj["low"].is<int>();
obj.is<int>("low");
```

Here too, the two statements are equivalent and produce the same executable.

Variant type and C++ types

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

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

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



More on arduinojson.org

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

Test if a key exists in an object

If you have an object and want to know whether a key exists in the object, 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 `containsKey()` can be avoided:

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

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

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

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

3.5 Parse a JSON array

The JSON document

We've seen how to parse a JSON object from a Yahoo Weather forecast; it's time to move up a notch by parsing an array of object. Indeed, the weather forecast comes in sequence: one object for each day.

Here is our example:

```
[
  {
    "item": {
      "forecast": {
        "date": "09 Nov 2017",
        "high": "53",
        "low": "38",
        "text": "Mostly Cloudy"
      }
    }
  },
  {
    "item": {
      "forecast": {
        "date": "10 Nov 2017",
        "high": "47",
        "low": "26",
        "text": "Breezy"
      }
    }
  },
  {
    "item": {
      "forecast": {
        "date": "11 Nov 2017",
        "high": "39",
        "low": "24",
        "text": "Partly Cloudy"
      }
    }
  }
]
```

Hum... that's not exactly what I expected, but alright...

So instead of just an array of `forecast` objects, Yahoo Weather returns a JSON document with four levels of nesting:

1. The root is an array of objects.
2. Each object contains a nested object named `item`.
3. Each `item` contains a nested object named `forecast`.
4. Each `forecast` contains the information we want: `date`, `high`, `low` and `text`.

This document is not as straightforward as one would hope but it's not that complicated either. Furthermore, it perfectly illustrates a problem that many ArduinoJson use encounter: the confusion between object and array.



Optimized cross-product

With Yahoo Weather, it's possible to pass an extra parameter to change the layout of the array to match our initial expectation. This parameter is `crossProduct=optimized`. However, if we use it, we lose the ability to limit the number of days in the forecast and we take the risk of having a response that is too big for our ATmega328. We could get along with that, as we'll see [in the case studies](#), but I want to keep things simple for your first contact with ArduinoJson.

Parse the array

You should now be familiar with the process:

1. Put the JSON document in memory.
2. Allocate the `JsonBuffer`.
3. Call `parseObject()` `parseArray()`.
4. Check the return value of `success()`.

Let's do it:

```

// Put the JSON input in memory (shortened)
char input[] = "[{\n\"item\":{\n\"forecast\":{\n\"date\":\n\"09 Nov 2017\\\", \"\\\"high...\";

// Allocate the JsonBuffer
const int capacity = JSON_ARRAY_SIZE(3)
                    + 6*JSON_OBJECT_SIZE(1)
                    + 3*JSON_OBJECT_SIZE(4);
StaticJsonBuffer<capacity> jb;

// Parse the JSON input
JsonArray& arr = jb.parseArray(input);

// Parse succeeded?
if (arr.success()) {
    // Yes! We can extract values.
} else {
    // No! The input may be invalid, or the JsonBuffer may be too small.
}

```

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


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

- There is one array of three elements: `JSON_ARRAY_SIZE(3)`
- In this array, there are three objects of one element: `3*JSON_OBJECT_SIZE(1)`
- In each object, there is one object (`item`) containing one element: `3*JSON_OBJECT_SIZE(1)`
- In each `item`, there is one object (`forecast`) containing four elements: `3*JSON_OBJECT_SIZE(4)`

The ArduinoJson Assistant

For complicated JSON documents, the expression to compute the capacity of the `JsonBuffer` becomes impossible to write by hand. Here, I did it so that you understand the process; but, in practice, we use a program to do this task.

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

 **ArduinoJson** [API Reference](#) [Manual](#) [Examples](#) [FAQ](#) [Assistant](#)

Fork me on GitHub

ArduinoJson Assistant

Input

```
[{"item":{"forecast":{"date":"09 Nov 2017","high":"53","low":"38","text":"Mostly Cloudy"}}}, {"item":{"forecast":{"date":"10 Nov 2017","high":"47","low":"26","text":"Breezy"}}}, {"item":{"forecast":{"date":"11 Nov 2017","high":"39","low":"24","text":"Partly Cloudy"}}}]
```

Examples: [OpenWeatherMap](#), [Weather Underground](#)

JsonBuffer size

Expression

```
JSON_ARRAY_SIZE(3) +  
6*JSON_OBJECT_SIZE(1) +  
3*JSON_OBJECT_SIZE(4)
```

Additional bytes for input duplication

```
188
```

Platform	Size
AVR 8-bit	432
ESP8266	592
Visual Studio x86	956
Visual Studio x64	1076

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

3.6 Extract values from an array

Unrolling the array

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

To get access to the forecast data, we need to unroll the nested objects. Here is the code to do it, step by step:

```
// Get the first element of the array
JsonObject& arr0 = arr[0];

// Get the `item` object inside this object
JsonObject& item0 = arr0["item"];

// Get the `forecast` object inside this object
JsonObject& forecast0 = item0["forecast"];
```

And we're back to the `JsonObject` with four elements: `date`, `low`, `high` and `text`. This subject was entirely covered in the previous selection, so there is no need to repeat.

Fortunately, it's possible to simplify the program above with just a single line:

```
// Get the first `forecast` object
JsonObject& forecast0 = arr[0]["item"]["forecast"];
```

Alternative syntaxes

It may not be obvious, but the two programs above use implicit casts. Indeed, the return value of the subscript operator (`[]`) is a `JsonVariant`; the `JsonVariant` is then implicitly converted to a `JsonObject&`.

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

```
auto arr0 = arr[0].as<JsonObject&>();
```

And there is also another form with `JsonArray::get<T>()`:

```
auto arr0 = arr.get<JsonObject>(0);
```

All of this should sound very familiar because as it's the same as objects.

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). It is the same if you use an index that is out of the range of the array.

Now is a good time to see what happens if a complete object is missing. For example:

```
// Get an object out of array's range
JsonObject& forecast666 = arr[666]["item"]["forecast"];
```

The index 666 doesn't exist in the array, so a special value is returned: `JsonObject::invalid()`. It's a special object that doesn't contain anything and whose `success()` method always returns false:

```
// Does the object exists?
if (!forecast666.success()) {
    // Of course not!
}
```

There are two special objects like this: `JsonArray::invalid()` and `JsonObject::invalid()`. They are just here to fill the hole when a `JsonArray` or a `JsonObject` is missing. Usually, your program doesn't have to deal with them directly, so you don't have to remember them.



The null-object design pattern

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

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

```
JsonObject& forecast0 = arr[0]["item"]["forecast"];
```

3.7 Inspect an unknown array

Our example was very straightforward because we knew that the JSON array had precisely three elements and we knew the content of these elements. In this section, we'll see what tools are available when the content of the array is not known.

Capacity of `JsonBuffer` for an unknown input

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

This constraint looks terrible at first, especially if you're a desktop or server application developer; but, once you think about it, it makes complete sense. Indeed, your program is going to run in a loop, always on the same hardware, with a known amount of memory. Having an elastic capacity would just produce a larger and slower program with no additional value.

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

Number of elements in an array

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

```
int count = arr.size();
```

As the name may be confusing, I insist that `JsonArray::size()` returns the number of elements, not the memory consumption. If you want to know how many bytes of memory are used, call `JsonBuffer::size()`:

```
int memoryUsed = jb.size();
```

Remark that `JsonObject` also has a `size()` method returning the number of key-value pairs, but it's rarely useful.

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& forecast = arr[i]["item"]["forecast"];
}
```

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

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

```
// Walk the JsonArray efficiently
for (JsonObject& elem : arr) {
    JsonObject& forecast = elem["item"]["forecast"];
}
```

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

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



When C++11 is not available

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

```
for (JsonArray::iterator it=arr.begin(); it!=arr.end(); ++it) {
    JsonObject& elem = *it;
}
```

Detect the type of the elements

We test the type of array elements the same way we did for object values. In short, we can either use `JsonVariant::is<T>()` or `JsonArray::is<T>()`.

Here is a code sample with all syntaxes:

```
// Is the first element an integer?
if (arr[0].is<int>()) {
    // We called JsonVariant::is<int>()
}

// Is the second element a float?
if (arr.is<float>(1)) {
    // We called JsonArray::is<float>(int)
}

// Same in a loop
for (JsonVariant& elem : arr) {
    // Is the current element an object?
    if (elem.is<JsonObject>()) {
        // We called JsonVariant::is<JsonObject>()
    }
}
```

3.8 The zero-copy mode

Definition

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

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

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

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



The jsmn library

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

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

An example

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

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

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

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

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

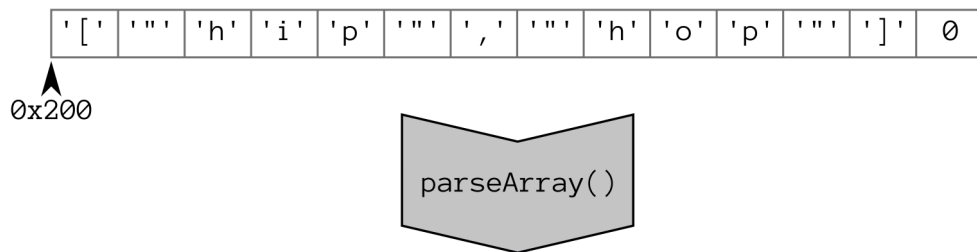
```
JsonArray& arr = jb.parseArray(input);

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

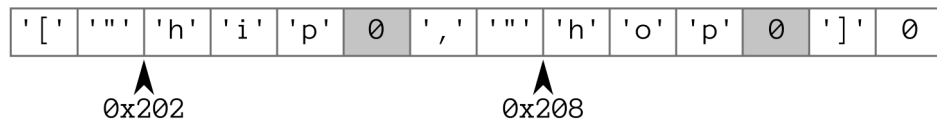
We naturally expect `hip` to be `"hip"` and not `"hip\\",\\"hop\\""`; that's why `ArduinoJson` adds a null-terminator after the first `p`. Similarly, we expect `hop` to be `"hop"` and not `"hop\\""`, so a second null-terminator is added.

The picture below summarizes this process.

Input buffer before parsing



Input buffer after parsing



How input buffer is modified

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

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

Input buffer must stay in memory

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

If a program dereferences the pointer after the destruction of the input buffer, it will create an undefined behavior (a UB, in the C++ jargon), which means that the code may work by accident, but will crash sooner or later.

Here is an example:


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

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

    // Parse input
    JsonArray& arr = jb.parseArray(input);

    // Save a pointer
    hip = arr[0];
}
// input is destructed now

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



Common cause of bugs

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

Always remember that, to use a `JsonArray` or a `JsonObject`, you must keep the `JsonBuffer` alive. In addition, when using the zero-copy mode, you must also keep the input buffer in memory.

3.9 Parse from read-only memory

The example

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

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

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

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

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

Duplication is required

As you saw in the previous section, in the zero-copy mode, ArduinoJson returns a pointer pointing inside the input buffer. We saw that it has to replace some characters of the input with null-terminators. But, with a read-only input, ArduinoJson cannot do that anymore; to return a null-terminated string, it needs to make copies of “hip” and “hop”.

Where do you think the copies would go? In the `JsonBuffer` of course!

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

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

The exact capacity required is:

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

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



Use the ArduinoJson Assistant

The ArduinoJson assistant also computes the number of bytes required for the duplication of the string. It's the field named "Additional bytes for input duplication."

ArduinoJson Assistant

API Reference Manual Examples FAQ Assistant

Fork me on GitHub

Input

["hip","hop"]

JsonBuffer size

Expression

JSON_ARRAY_SIZE(2)

Additional bytes for input duplication

8

Platform	Size
AVR 8-bit	28
ESP8266	40

Practice

Apart from the capacity of the JsonBuffer, we don't need to change anything to the program. Here is the complete hip-hop example with read-only input:

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

// Allocate the JsonBuffer
const int capacity = JSON_ARRAY_SIZE(2) + 8;
StaticJsonBuffer<capacity> jb;

// Parse the JSON input.
JsonArray& arr = jb.parseArray(input);

// Extract the two strings.
```

```
const char* hip = arr[0];
const char* hop = arr[1];

// How much memory is used?
int memoryUsed = jb.size();
```

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

If you compile this program on an ATmega328, the variable `memoryUsed` will contains 28, as the ArduinoJson Assitant predicted.

Other types of read-only input

`const char*` is not the only type you can use. It's possible to use a `String`:

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

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

The simplest is to use the `F()` macro:

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

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

3.10 Parse from stream

In the Arduino jargon, a stream is a volatile source of data, like a serial port. As opposed to a memory buffer, which allows reading any bytes at any location, a stream only allows to read one byte at a time and cannot go back.

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

Library	Class	Well known instances
Core	<code>HardwareSerial</code>	<code>Serial</code> , <code>Serial1</code> ...
ESP8266 FS	<code>File</code>	
Ethernet	<code>EthernetClient</code>	
Ethernet	<code>EthernetUDP</code>	
GSM	<code>GSMClient</code>	
SD	<code>File</code>	
SoftwareSerial	<code>SoftwareSerial</code>	
Wifi	<code>WifiClient</code>	
Wire	<code>TwoWire</code>	<code>Wire</code>



`std::istream`

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

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

Parse from a file

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

The program will just read the file and print the content of the weather forecast for each day.

Here is the relevant part of the code:

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

// Parse directly from file
JsonArray& arr = jb.parseArray(file);

// Loop through all element of the array
for (JsonObject& elem : arr) {
    // Extract the forecast object
    JsonObject& forecast = elem["item"]["forecast"];

    // Print weather
    Serial.println(forecast["date"].as<char*>());
    Serial.println(forecast["text"].as<char*>());
    Serial.println(forecast["high"].as<int>());
    Serial.println(forecast["low"].as<int>());
}
```

A few things to note:

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

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

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

Parse from an HTTP response

Now is the time to parse the real data coming from Yahoo Weather server.

Yahoo services use a custom language named “YQL” to perform a query. Carefully crafting the query allows to retrieve only the information we need, and therefore reduces the work of the microcontroller.

In our case the query is:

```
select item.forecast.date,  
       item.forecast.text,  
       item.forecast.low,  
       item.forecast.high  
from weather.forecast(3)  
where woeid=2459115
```

This query asks for the weather forecast of the city of New York (woeid=2459115) and limits the results to three days (weather.forecast(3)). As we don't need all forecast data, we only select relevant columns: date, text, low and high.

The YQL query is passed as an HTTP query parameters, here is URL we need to fetch:

```
http://query.yahooapis.com/v1/public/yql?q=select%20item.forecast.date%2Citem...
```

The HTTP request we need to send is:

```
GET http://query.yahooapis.com/v1/public/yql?q=select%20...&format=json HTTP/1.0  
Host: query.yahooapis.com  
Connection: close
```

The HTTP response we receive looks like:

```
HTTP/1.0 200 OK  
Content-Type: application/json; charset=utf-8  
Date: Tue, 14 Nov 2017 09:57:39 GMT  
  
{"query":{"count":3,"created":"2017-11-14T09:57:39Z","lang":"en-US","results"...
```

The JSON document in the body looks like this:

```
{  
  "query": {  
    "count": 3,  
    "created": "2017-11-14T09:57:39Z",  
    "lang": "en-US",  
    "results": {  
      "channel": [  
        {  
          "item": {  
            "forecast": {
```

```

        "date": "14 Nov 2017",
        "high": "46",
        "low": "36",
        "text": "Partly Cloudy"
    }
}
},
{
    "item": {
        "forecast": {
            "date": "15 Nov 2017",
            "high": "47",
            "low": "38",
            "text": "Mostly Cloudy"
        }
    }
},
{
    "item": {
        "forecast": {
            "date": "16 Nov 2017",
            "high": "52",
            "low": "43",
            "text": "Partly Cloudy"
        }
    }
}
]
}
}

```

As the class `EthernetClient` also derives from `Stream`, we can pass it directly to `parseObject()`, just like we did with `File`.

The following program performs the HTTP request and displays the result in the console:


```

// Connect to HTTP server
EthernetClient client;
client.setTimeout(10000);
client.connect("query.yahooapis.com", 80);

// Send HTTP request (shortened)
client.println("GET /v1/public/yql?q=select%20item.forecast.date%2Citem.fo...");
client.println("Host: query.yahooapis.com");
client.println("Connection: close");
client.println();

// Skip response headers
char endOfHeaders[] = "\r\n\r\n";
client.find(endOfHeaders);

// Allocate JsonBuffer
const size_t capacity = JSON_ARRAY_SIZE(3)
    + 8*JSON_OBJECT_SIZE(1)
    + 4*JSON_OBJECT_SIZE(4)
    + 300;
StaticJsonBuffer<capacity> jsonBuffer;

// Parse response
JsonObject& root = jsonBuffer.parseObject(client);

// Extract the array "query.results.channel"
JsonArray& results = root["query"]["results"]["channel"];

// Loop through the element of the array
for (JsonObject& result : results) {
    // Extract the object "item.forecast"
    JsonObject& forecast = result["item"]["forecast"];

    // Print the values to the Serial
    Serial.println(forecast["date"].as<char*>());
    Serial.println(forecast["text"].as<char*>());
    Serial.println(forecast["high"].as<int>());
    Serial.println(forecast["low"].as<int>());
}

```

A few remarks:

1. I used HTTP 1.0 instead of 1.1 to avoid [Chunked transfer encoding](#).

2. We're not interested in the response's headers, so we skip them using `Stream::find()`, placing the reading cursor right at the beginning of the JSON document.
3. `Stream::find()` takes a `char*` instead of a `const char*`, that's why we need to declare `endOfHeaders`.
4. As usual, I used the ArduinoJson Assistant to compute the capacity of the `JsonBuffer`.

You can find the complete source code of this example in the folder `YahooWeather` in the zip file. We will see two other weather services in the [case studies](#).

4. Serialize with ArduinoJson



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

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

4.1 The example of this chapter

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

The previous chapter revolved around weather forecast for the city of New York. We'll use a very different example for this chapter: pushing data to [Adafruit IO](#). Adafruit IO is a cloud storage for IoT data. They have a free plan with the following restrictions:

- 30 data points per minutes
- 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.



The Adafruit IO front page

4.2 Create an object

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 it has no nested object or array, and it contains the following piece of information:

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

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

Allocate the JsonBuffer

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

We need to compute the capacity of the `JsonBuffer`. As the JSON document is simple, we can do the computation manually. We just have one object with no nested values, so the capacity is `JSON_OBJECT_SIZE(3)`. Remember that you can use ArduinoJson Assistant when the JSON document is more complex.

It's a fairly small `JsonBuffer`, which fits in the stack on any microcontroller, so we can use a `StaticJsonBuffer`.

Here is the code:

```
const int capacity = JSON_OBJECT_SIZE(3);  
StaticJsonBuffer<capacity> jb;
```

Create the object

It's not possible to directly instantiate a `JsonObject`, we'll see why [in the next chapter](#). Instead, we need to ask the `JsonBuffer` to create one for us:

```
// Create a JsonObject  
JsonObject& obj = jb.createObject();
```

This statement creates an empty object; its memory usage is currently `JSON_OBJECT_SIZE(0)`

As the object is located inside the `JsonBuffer`, we receive a reference to it. We saw the same thing with `parseObject()` [in previous chapter](#).



The factory design-pattern

`JsonObject` cannot be constructed directly, but `JsonBuffer` provides a method to create an object.

It is an implementation of the “factory” [design pattern](#): `JsonBuffer` is a factory of `JsonObject`.

Add the values

Adding values to a `JsonObject` is very similar to reading them.

There are several syntaxes, but the simplest is to use the subscript operator (`[]`):

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

The object’s memory usage is now `JSON_OBJECT_SIZE(3)`, meaning that the `JsonBuffer` is full. When the `JsonBuffer` is full, you cannot add more values to the object, so don’t forget to increase the capacity if you need.

Second syntax

With the syntax presented above, it’s not possible to tell if the insertion succeeded or failed. Let’s see another syntax:

```
obj.set("value", 42);  
obj.set("lat", 48.748010);  
obj.set("lon", 2.293491);
```

The executable generated by the compiler is the same as with the previous syntax, except that you can check if the insertion succeeded. You can check the result of `JsonObject::set()` which will be `true` for success or `false` for failure. Again, insertion fails if the `JsonBuffer` is full.

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

Third syntax

The syntax we just saw used `JsonObject::set()`. We can combine both syntax by using `JsonVariant::set()`:

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

Again, the compiled executable is the same, and you can check the return value too.

Replace values

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

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

It doesn't require a new allocation in the `JsonBuffer`, so if the first insertion succeeds, the second will succeed too.

Remove values

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

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



Code smell

In practice, this problem only happens in programs that use `ArduinoJson` to store the state of the application, which is not what `ArduinoJson` is for. Trying to optimize this use-case would inevitably impact the size and speed of `ArduinoJson`.

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

4.3 Create an array

The example

Our next step will be to construct an array containing two objects:

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

Allocate the `JsonBuffer`

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

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

Here is the code:

```
const int capacity = JSON_ARRAY_SIZE(2) + 2*JSON_OBJECT_SIZE(2);
StaticJsonBuffer<capacity> jb;
```

Create the array

We create arrays the same way we create objects, by using the `JsonBuffer` as a factory:

```
JsonArray& arr = jb.createArray();
```

Add values

To add the nested objects to the array, we could create the two objects and add them to the array like that:


```
JsonObject& obj1 = jb.createObject();  
obj1["key"] = "a1";  
obj1["value"] = analogRead(A1);  
  
JsonObject& obj2 = jb.createObject();  
obj2["key"] = "a2";  
obj2["value"] = analogRead(A2);  
  
arr.add(obj1);  
arr.add(obj2);
```

`JsonArray::add()` adds a new value at the end of the array. In this case, we added two `JsonObject`s, but you can use any value that a `JsonVariant` handles: `int`, `float`...

This technique is one way to create an array of object, but it is not the best way. Instead, the recommended technique is the following:

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

Thanks to this technique, the program is shorter, faster, and more readable.

Replace values

As for objects, it's possible to replace values in arrays using either `JsonArray::operator[]` or `JsonArray::set()`:

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

Replacing the value doesn't require a new allocation in the `JsonBuffer`. However, if there was memory hold by the previous value, for example, a `JsonObject`, this memory is not released. Doing so would require counting references to the nested `JsonObject`, which `ArduinoJson` does not.

Remove values

As for objects, you can delete a slot of the array, by using `JsonArray::remove()`:

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

As described [in the previous section](#), `remove()` doesn't release the memory from the `JsonBuffer`. You should never call this function in a loop.

Add null

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

The first special value is `null`, which is a legal token in a JSON document. In `ArduinoJson`, it is a string whose address is zero:

```
// adds "null"
arr.add("null");

// add null
arr.add((char*)0);
```

The program above produces the following JSON document:

```
[
  "null",
  null
]
```

Add pre-formatted JSON

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

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

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

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

The program above produces the following JSON document:

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

Since version 5.13, the `RawJson()` function also supports `String` and `Flash` strings, and it duplicates the string when needed.

4.4 Serialize to memory

We saw how to construct an array, and 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 may now start to see, I don't like using dynamic memory allocation. Instead, we'd use a good old `char[]`:

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

Minified JSON

Suppose we're still using our previous example for `JsonArray`, if we want to produce a JSON document out of it, we just need to call `JsonArray::printTo()`:

```
// Produce a minified JSON document  
arr.printTo(output);
```

Now the string output contains:

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

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

Specify (or not) the size of the output buffer

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

```
arr.printTo(output, sizeof(output));
```

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

Of course, this only work because `output` is an array. If it were a `char*`, we would have had to specify the size.

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 just need to use `prettyPrintTo()` instead of `printTo()`:

```
// Produce a prettified JSON document  
arr.prettyPrintTo(output);
```

Here is the output:

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

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

Compute the length

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

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

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

```
// Compute the length of the minified JSON document  
int len1 = arr.measureLength();  
  
// Compute the length of the prettified JSON document  
int len2 = arr.measurePrettyLength();
```

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

By the way, `printTo()` and `prettyPrintTo()` return the number of bytes written. Their return values are the same as `measureLength()` and `measurePrettyLength()`, except if the output buffer was too small.



Avoid prettified documents

The sizes in the example above are 73 and 110. In this case, the prettified version is only 50% bigger, because the document is simple. But, in most case, the ratio is way above 100%.

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

Serialize to a String

The functions `printTo()` and `prettyPrintTo()` have overloads taking a `String`:

```
String output = "JSON = ";
```

```
arr.printTo(output);
```

The behavior is slightly different as the JSON document is appended to end the `String`. The snippet above sets the content of the `output` variable to:

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

Cast a JsonVariant to a String

You should remember from the chapter on deserialization that a `JsonVariant` have to be casted to the type we want to read.

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

We could rewrite the previous example like this:

```
// Wrap the JsonArray in a JsonVariant
```

```
JsonVariant v = arr;
```

```
// Cast the JsonVariant to a string
```

```
String output = "toto" + v.as<String>();
```

Unfortunately, this trick is only available on `JsonVariant`; you cannot do the same with `JsonArray` and `JsonObject` (unless you put them in a `JsonVariant`). Furthermore, this technique only produces a minified document.

4.5 Serialize to stream

What's an output stream?

In the previous section, we saw how to serialize an array or an object. For now, every JSON document we produced remained in memory, but that's usually not what we want.

In many situations, it's possible to send the JSON document directly to its destination (whether it's a file, a serial port, or a network connection) without any copy in RAM.

We saw in the previous chapter what an input stream is, and that Arduino represents this concept with the class `Stream`.

Similar to input streams, we also have “output streams,” which are a volatile sink of bytes. We can write to an output stream, but we cannot read. In the Arduino land, an output stream is materialized by the class `Print`.

Here are examples of classes derived from `Print`:

Library	Class	Well known instances
Core	<code>HardwareSerial</code>	<code>Serial</code> , <code>Serial1</code> ...
ESP8266 FS	<code>File</code>	
Ethernet	<code>EthernetClient</code>	
Ethernet	<code>EthernetUDP</code>	
GSM	<code>GSMClient</code>	
LiquidCrystal	<code>LiquidCrystal</code>	
SD	<code>File</code>	
SoftwareSerial	<code>SoftwareSerial</code>	
Wifi	<code>WifiClient</code>	
Wire	<code>TwoWire</code>	<code>Wire</code>



`std::ostream`

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

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

Serialize to Serial

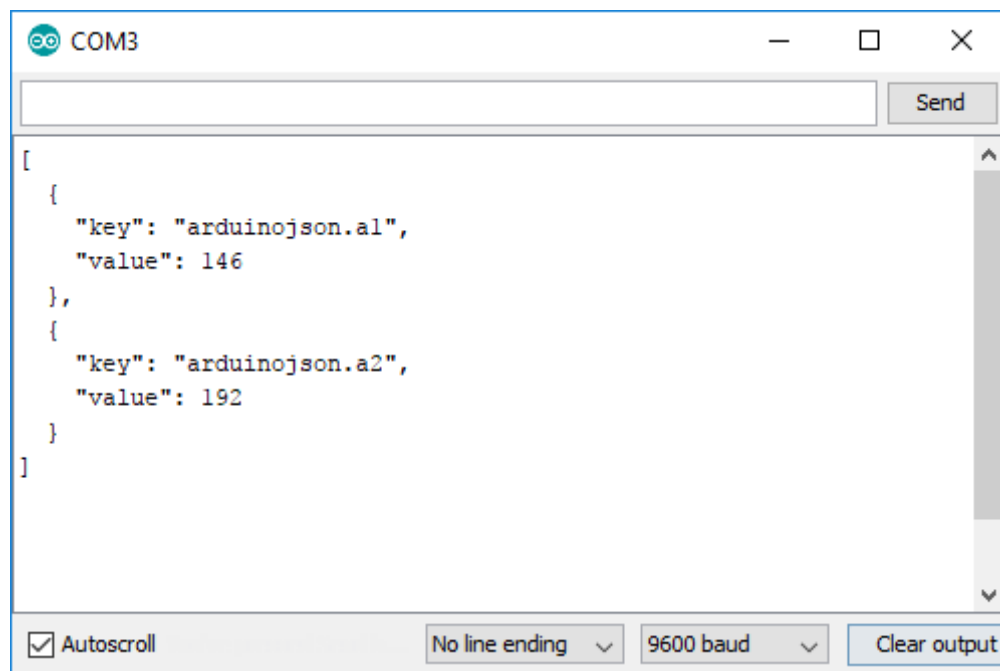
The most famous implementation of `Print` is `HardwareSerial`, which is the class of `Serial`. To serialize a `JsonArray` or a `JsonObject` to the serial port of your Arduino, just pass `Serial` to `printTo()`:

```
// Print a minified version to the serial port
arr.printTo(Serial);

// Same with a prettified version
arr.prettyPrintTo(Serial);
```

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

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



Result of prettyPrintTo() in Serial Monitor

Serialize to a file

In the exact same way, we can use a File instance as the target of printTo() and prettyPrintTo(). 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
arr.prettyPrintTo(file);
```

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

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

Serialize to an HTTP request

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

To do that, we need to send the following JSON document:

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

This document contains the values to add to our two feeds a1 and a2.

We will send this document in the body of the following HTTP request:

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

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

Here is the program:

```
// Allocate JsonBuffer
const int capacity = JSON_ARRAY_SIZE(2) + 4 * JSON_OBJECT_SIZE(2);
StaticJsonBuffer<capacity> jb;

// Create JsonObject
JsonObject &root = jb.createObject();

// Add location
JsonObject &location = root.createNestedObject("location");
location["lat"] = 48.748010;
location["lon"] = 2.293491;

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

// Connect to the HTTP server
EthernetClient client;
client.setTimeout(10000);
client.connect("io.adafruit.com", 80);

// Send HTTP request
client.println("POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.1");
client.println("Host: io.adafruit.com");
client.println("Connection: close");
client.print("Content-Length: ");
client.println(root.measureLength());
client.println("Content-Type: application/json");
client.println("X-AIO-Key: baf4f21a32f6438eb82f83c3eed3f3b3");
client.println();
root.printTo(client);
```

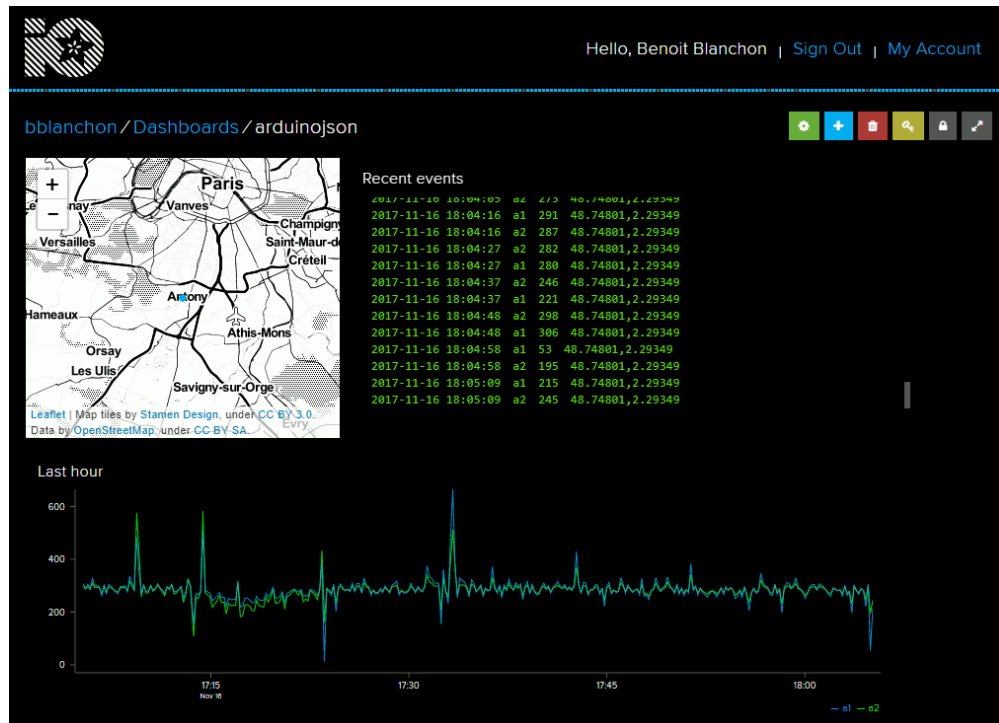
You can find the complete source code of this example in the folder `AdafruitIo` in the zip file.

If you want to reproduce this example, you need to follow these steps:

1. Create an account on Adafruit IO (a free tier is sufficient).
2. Create a feed group named `arduinojson`.

3. In that group, create two feeds a1 and a2.
4. Replace the username (bblanchon here) with yours.
5. Replace the AIO key (baf4f21a32f6438eb82f83c3eed3f3b3 here) with yours.

Below is a picture of a dashboard showing the data from this example.



The measurement as seen in Adafruit IO

4.6 Duplication of strings

When you add a string value to a `JsonArray` or a `JsonObject`, `ArduinoJson` either stores a pointer or a copy of the string depending on its type. If the string is a `const char*`, it stores a pointer; otherwise, it makes a copy.

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

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

The table above reflects the new rules that are in place since `ArduinoJson 5.13`; on older versions `char*` were stored with a pointer, which caused surprising effects.

An example

Compare this program:

```
// Create the array ["hello","world"]
JsonArray& arr = jb.createArray();
arr.add("hello");
arr.add("world");

// Print the memory usage
Serial.println(jb.size());
```

with the following:

```
// Create the array ["hello","world"]
JsonArray& arr = jb.createArray();
arr.add(String("hello"));
arr.add(String("world"));

// Print the memory usage
Serial.println(jb.size());
```

They both produce the same JSON document, but the second one require much more memory because `ArduinoJson` had to make copies. If you run these programs on an `ATmega328`, you'll see 20 for the first one and 32 for the second.

Copy only occurs when adding values

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

Here is an example:

```
JsonObject& obj = jb.createObject();

// The following line produces a copy of "hello"
obj[String("hello")] = "world";

// The following line produces no copy
const char* world = obj[String("hello")];
```

Why copying Flash strings?

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

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

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

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

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

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

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

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

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



Avoid Flash string with ArduinoJson

Storing strings in Flash is a great way to reduce RAM usage, but remember that they are copied into the `JsonBuffer`.

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

RawJson()

We saw [earlier in this chapter](#) that the `RawJson()` function marks strings as JSON segments that should not be treated as string values.

Since ArduinoJson 5.13, `RawJson()` supports all the string types (`char*`, `const char*`, `String` and `const __FlashStringHelper*`) and obeys to the rules stated in the table above.

5. 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++: 42 Specific Ways to Improve Your Use of C++11 and C++14*

5.1 Why JsonBuffer?

On your first contact with ArduinoJson, I'm sure you had this thought: "What is this curious `JsonBuffer` and why do we need it?" I'll try to answer both questions in this section.

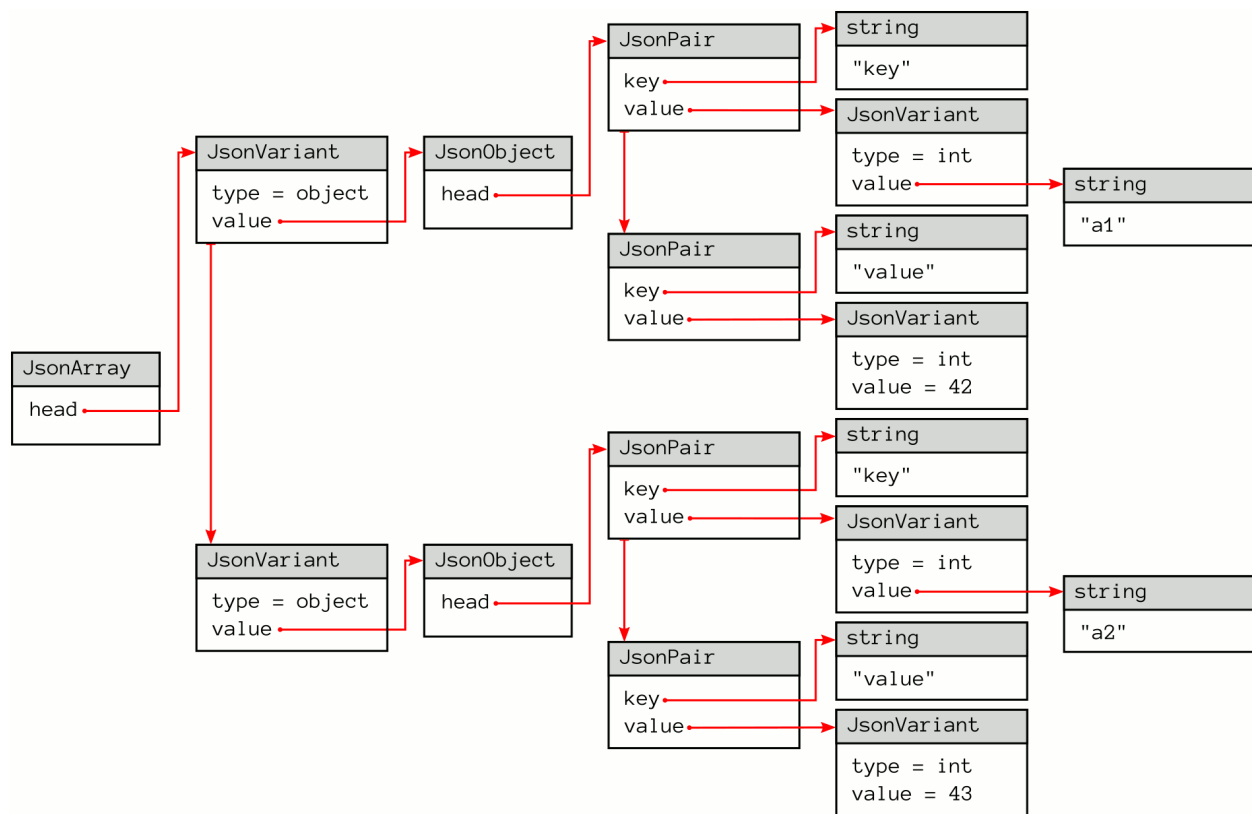
Memory representation

To illustrate this section, we'll take the simple JSON document from the previous chapter:

```
[
  {
    "key": "a1",
    "value": 42
  },
  {
    "key": "a2",
    "value": 43
  }
]
```

It's an array of two elements. Both elements are objects. Each object has two values: a string named `key` and an integer named `value`.

This document is a fairly simple; however its in-memory representation can be quite complex, as you can see in the diagram below.



In-memory representation of a simple array

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

A `JSONArray` is a linked list of `JsonVariant`, that's why it contains a pointer to the first variant which is then linked to the second. The same logic is implemented in `JsonObject`.

Dynamic memory

Our JSON document is very simple, yet it requires 19 objects in memory, as we see in the diagram.

If you were to implement your own JSON library, you would probably allocate each object using `new`. That would be the most natural technique in a program written in Java or C#. In fact, this is how most JSON libraries are implemented, including `aJson`.

Unfortunately, dynamic memory allocation has a cost:

1. Each allocation produces a small memory overhead, which is not negligible when the objects are small like ours.
2. Each allocation and each deallocation requires a significant housekeeping work from the microcontroller.
3. Repeated allocations produce “memory fragmentation”, as we saw [in the C++ course](#).

This approach is not suitable for an embedded program. Nineteen allocations followed by nineteen deallocations is an unacceptable waste of CPU cycles. This is what makes ArduinoJson different from other libraries: it manages to do everything with just one allocation and one deallocation.

Memory pool

ArduinoJson reduces the allocation/deallocation work to the minimum by using a “memory pool.” Instead of making 19 small allocations, it makes a big one to create the memory pool. Then it creates the 19 objects inside the pool. The cost of allocation in the pool is almost free (just a few CPU cycles).

ArduinoJson also takes a radical choice concerning the deallocation in the pool: it is simply not supported. Yes, you read it right. It’s not possible to release memory inside the pool. However, it’s possible to destroy the memory pool, thereby freeing the memory of the 19 objects.

Since it’s not possible to delete an object inside the pool, none of the objects have a destructor. Therefore the destruction of the memory pool is very fast.

StaticJsonBuffer and DynamicJsonBuffer are the two types of memory pool available in ArduinoJson. They both derive from the abstract class JsonBuffer.



Never reuse a JsonBuffer

Because JsonBuffer cannot deallocate memory, it only grows during its lifetime. It is crucial not to reuse a JsonBuffer as it would continue to grow until it’s full.

Never use global JsonBuffers. [The case studies](#) show several good examples, none of them uses a global JsonBuffer.

Strengths and weaknesses

The advantages of this implementation are:

1. allocations are super fast,
2. deallocations are even faster,
3. no fragmentation of the heap,
4. minimal code size.

The drawbacks are:

1. memory pool is a new concept,
2. object in the pool cannot be freed.



Remove values from object and array

Because it's not possible to delete an object inside the pool, the functions `JsonArray::remove()` and `JsonObject::remove()` leak memory. Do not use them in a loop; otherwise, the `JsonBuffer` would be full after a few iterations.

5.2 Inside StaticJsonBuffer

The `StaticJsonBuffer` is the simplest implementation of `JsonBuffer`.

Fixed capacity

A `StaticJsonBuffer` has a fixed capacity; it cannot grow at run-time. When it's full (meaning when it's size reached its capacity), a `StaticJsonBuffer` rejects all further allocation, just like `malloc()` would do if the heap was full.

As a consequence, you need to determine the appropriate capacity at the time you create a `StaticJsonBuffer`. As said before, you can use the ArduinoJson Assistant on arduinojson.org to determine the right capacity for your project.

Compile-time determination

The capacity must be set at compile-time as a template parameter. As we said, you cannot use a variable to specify the capacity. Instead, you need to use a constant (`const` or `constexpr` in C++).

```
int capacity = 42;  
const int capacity = 42;
```



Compile-time vs run-time

“Compile-time” refers to the work done by the compiler when it generates the binary executable for the program. “Run-time” refers to the work done by the program when it runs on the target platform.

Experienced C++ programmers try to do as much work as possible at compile-time, to improve the performance at run-time.

Stack memory

Because its capacity is constant, a `StaticJsonBuffer` can be allocated in the stack, and it is the recommended usage.



Location of a `StaticJsonBuffer` in RAM

As said in the C++ course, allocation in the stack is very fast, because it doesn't require to look for available memory; the compiler does all the work in advance. Creating and destructing a `StaticJsonBuffer` in the stack costs a handful of CPU cycles; exactly like a local variable.

Here is a program that allocate a `StaticJsonBuffer` on the stack:

```
StaticJsonBuffer<capacity> jb;
```



Prefer stack

As the cost of allocation is virtually zero, seasoned C++ programmers try to put everything in the stack.

Limitation

Of course, the stack size cannot exceed the amount of physical memory installed on the board. On top of that, many platforms put a limit on the stack size. See the table below for a list of popular platforms and limits:

Platform	Available RAM	Default stack size
ESP8266	96KB	4KB
ESP32	520KB	8KB
ATmega328	2KB	unlimited
MSP430	<=10KB	unlimited

As a general rule, platforms with a very small amount of RAM put no limit on stack size. On a computer program, the stack size is typically limited to 1MB.

On platforms that limit the size of the stack, if the program allocates more memory than available, it crashes. Therefore, on these platforms, you cannot use a huge `StaticJsonBuffer`, you need to switch to heap memory. The stack size can always be modified, but it's easier to switch to `DynamicJsonBuffer`, even if the performance is a little inferior.



Why limit the stack size?

Limiting the stack size allows detecting bugs, like infinite recursions.

Indeed, each time a function is called, its arguments and the return address are pushed to the stack. This copy is repeated each time there is a recursion. If the recursion is infinite the stack grows until it reaches the limit, and boom!

Other usages

It's possible to allocate a `StaticJsonBuffer` in the heap by using `new`, but I strongly recommend against it, because you would lose the RAII feature. Instead, it's safer to use a `DynamicJsonBuffer`.

It's possible to use a `StaticJsonBuffer` as a member of an object, but it should not be a way to hide a global `JsonBuffer`. We'll see an implementation of this technique in the [case studies](#)

Implementation

The implementation of `StaticJsonBuffer` is surprisingly simple: it is just an array of bytes and an integer to keep track of the current usage. Allocating in the pool is just a matter of increasing the integer.

Here is a simplified definition:

```
template <int capacity>
class StaticJsonBuffer : public JsonBuffer {
    int size = 0;
    char data[capacity];

public:
    virtual void *alloc(int n) {
        // Verify that there is enough space
        if (size + n <= capacity)
            return nullptr;

        // Take the address of the next unused byte
        void *p = &data[size];

        // Increment size
        size += n;

        return p;
    }
};
```

Step by step

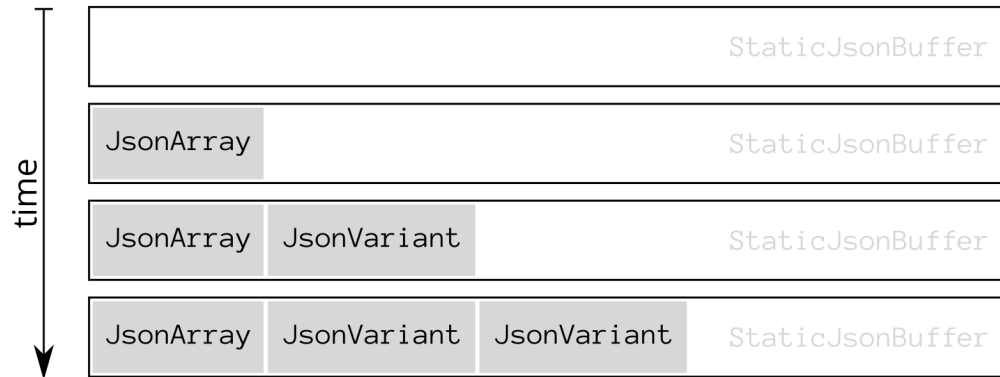
Let's see `StaticJsonBuffer` in action, imagine we have the following code:

```

StaticJsonBuffer<200> jb;           // empty
JsonArray& arr = jb.createObject(); // JsonArray
arr.add(42);                        // JsonArray + JsonVariant
arr.add("hello");                   // JsonArray + JsonVariant + JsonVariant

```

The comments show the content of the `StaticJsonBuffer` after executing each line. The picture below shows how the buffer fills over time:



Code size

This very simple mechanism leads to a very small binary executable. If you struggle to make your program fit in the little Flash on the board, make sure you use a `StaticJsonBuffer`.

You can go a step further by removing every dynamic allocation from your program; it allows the compiler (more exactly the linker) to drop the functions performing the memory management.

5.3 Inside DynamicJsonBuffer

Compared to `StaticJsonBuffer`, `DynamicJsonBuffer` brings two new features to the table: dynamic memory allocation and variable capacity.

Chunks

A `DynamicJsonBuffer` allocates its memory in “chunks.” A chunk is just a big byte array. The buffer begins with only one chunk, and creates more when needed. All chunks are allocated in the heap. `DynamicJsonBuffer`’s destructor destroys all the chunks at once.



Location of a `DynamicJsonBuffer` in RAM

The `DynamicJsonBuffer` uses the memory of the chunk to make allocations for the calling program. In a sense, each chunk behaves like a `StaticJsonBuffer`. If there is not enough memory in the last chunk, the `DynamicJsonBuffer` creates a new one. Each time it creates a chunk, it doubles the size of the previous one. The constructor’s argument specifies the size of the first chunk.

Performance

The cost of a `DynamicJsonBuffer` is directly related to the number of allocation it makes.

If there is only one chunk, it only adds the cost of one allocation and one deallocation, compared to a `StaticJsonBuffer`. For a big chunk that contains a lot of objects, the cost of the allocation and deallocation is negligible.



Specify the capacity to the constructor

The argument to the constructor of `DynamicJsonBuffer` is optional, but it’s not a reason to ignore it! This argument defines the size of the first chunk, so it allows to have only one chunk. Keeping the `DynamicJsonBuffer` to one chunk improves program speed and memory usage, making it very close to a `StaticJsonBuffer`.

Step by step

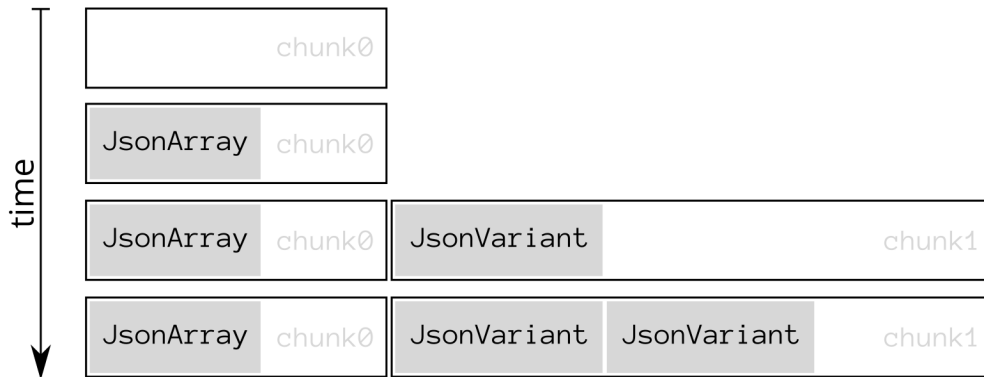
Let’s see the `DynamicJsonBuffer` in action, imagine we have the following code:


```

DynamicJsonBuffer jb(40)           // empty
JsonArray& arr = jb.createObject(); // JsonArray
arr.add(42);                        // JsonArray + JsonVariant
arr.add("hello");                   // JsonArray + JsonVariant + JsonVariant

```

The comments show the content of the DynamicJsonBuffer after executing each line. The picture below shows how it fills over time:



In this example, the size of the first chunk has been reduced to show how the allocation of new chunk works. In a real case, the allocation of the second chunk would happen after more allocations.

As you can see, there is a lot of space lost at the end of each chunk. To avoid this waste, you should try to use only one chunk, by specifying the right capacity. As usual, use the ArduinoJson Assistant on arduinojson.org to determine the capacity.

Comparison with StaticJsonBuffer

To summarize, here is a feature comparison:

	StaticJsonBuffer	DynamicJsonBuffer
Location	stack	heap
Construction cost	near 0	one <code>malloc()</code> per chunk
Destruction cost	near 0	one <code>free()</code> per chunk
Cost of allocation	near 0	near 0, unless it creates a new chunk
Capacity	fixed	elastic
Specified in	template parameter	constructor parameter
Code size	near 0	pulls <code>malloc()</code> and <code>free()</code>

How to choose?

Always start with a StaticJsonBuffer as it's simpler and faster. Switch to a DynamicJsonBuffer only when the StaticJsonBuffer becomes too big.

Refer to [the table in the previous section](#) to know when a `StaticJsonBuffer` is too big. As a general rule, switch to `DynamicJsonBuffer` as soon as you need more than half of the stack capacity. For example, on the ESP8266, the stack is limited to 4KB, so the biggest `StaticJsonBuffer` you should use is 2KB.

On platforms that put no limit on the size of the stack, always use a `StaticJsonBuffer`. It doesn't make sense to use a `DynamicJsonBuffer` there.

5.4 Inside JsonArray

Implementation

JsonArray is just a collection of JsonVariants; it is implemented as a linked-list. Each node of the linked-list is allocated in the JsonBuffer. JsonArray holds a pointer to its JsonBuffer, allowing it to allocate more nodes when needed.

Here is a simplified definition:

```
struct JsonArrayNode {  
    JsonVariant value;  
    JsonArrayNode *next;  
};  
  
struct JsonArray {  
    JsonArrayNode *head;  
    JsonBuffer *buffer;  
};
```

Creating a JsonArray

JsonArray's constructor is private, so it's not possible to simply declare a JsonArray and use it. Instead, you must call `JsonBuffer::createArray()` which returns a reference to a JsonArray:

```
JsonArray arr;  
StaticJsonBuffer<200> jb;  
JsonArray& arr = jb.createArray();
```

See chapter [“Serialize with ArduinoJson”](#) for a tutorial.

Parsing a JsonArray

To parse a JsonArray, you need to use the JsonBuffer too:

```
StaticJsonBuffer<200> jb;  
JsonArray& arr = jb.parseArray(input);
```

See chapter [“Deserialize with ArduinoJson”](#) for a tutorial.

Invalid

When `JsonBuffer::createArray()` and `JsonBuffer::parseArray()` fail, they return a reference to a special instance of `JsonArray`. You can get this reference by calling `JsonArray::invalid()`.

This special instance implements the null-object pattern. The null-object has no behavior, so all its methods do nothing. It simplifies the program because it doesn't need to check that the return value is not null. See chapter [“Deserialize with ArduinoJson”](#) for an example.

You'll probably never call `JsonArray::invalid()` in your program, but `ArduinoJson` uses this function every time it cannot return a reference to a `JsonArray`.

Copying a JsonArray

`JsonArray`'s copy-constructor and copy-assignment are private, so it's not possible to copy a `JsonArray`. If you need to pass a `JsonArray` to a function, you must use a reference.

I've never seen the case, but if you want to make a complete copy of a `JsonArray`, you need to iterate through all elements of the first and add each element to the second.

JsonArray as a generic container

I've seen several programs using a `JsonArray` as a generic container. It is a bad idea for three reasons:

1. The versatility of `JsonVariant` causes a significant overhead in speed and memory usage, compared to a custom data structure.
2. `JsonArray::remove()` can remove a node from the list but doesn't release the memory in the `JsonBuffer`, causing a memory leak.
3. The lookup in a `JsonArray` has a $O(n)$ complexity, so the performance is low compared to a simple array.

In the [case studies](#), we'll use `ArduinoJson` to store the configuration of a program; as you'll see, there is no need to use a global `JsonArray` as a generic container.

Methods

A detailed list of methods and overloads is available in the [“API Reference” on arduinojson.org](#). Here is a summary:

- `add(value)` adds a new `JsonVariant` to the end.
- `begin()` and `end()` return STL-style iterators.
- `copyFrom(array)` fills the `JsonArray` from the content of the specified array.

- `copyTo(array)` fills the specified array from the content of the `JsonArray`.
- `createNestedArray()` adds a new `JsonArray` to the end, and returns a reference.
- `createNestedObject()` adds a new `JsonObject` to the end, and returns a reference.
- `get<T>(index)` converts the `JsonVariant` at the specified index to the type `T`, and returns a default value (like `0`, `0.0` or `nullptr`) if the type is incompatible.
- `invalid()` returns a reference to the null-object.
- `is<T>(index)` tells if the `JsonVariant` at specified index has the type `T`.
- `measureLength()` computes the size of the minified JSON document.
- `measurePrettyLength()` computes the size of the prettified JSON document.
- `operator[] (index)` provides an alternative syntax for `set()` and `get()`.
- `prettyPrintTo(destination)` produces a prettified JSON document, and returns the number of bytes written.
- `printTo(destination)` produces a minified JSON document, and returns the number of bytes written.
- `remove(index)` removes the `JsonVariant` but doesn't release the memory in the `JsonBuffer`.
- `set(index, value)` replaces the `JsonVariant` at specified index.
- `size()` returns the number of elements in the `JsonArray`.
- `success()` tells if the `JsonArray` is valid (i.e. not the null-object).

In the methods above:

- `index` is a integer
- `value's type or T` can be:
 - `bool`
 - `char / short / int / long / long long` (all signed or unsigned)
 - `float / double`
 - `String / std::string`
 - `JsonArray / JsonObject`
 - `char* / const char* / char[]`
 - `const __FlashStringHelper*`
 - `RawJson`
 - `JsonVariant`
- `destination's type` can be:
 - `char[]`
 - `char*` (plus an integer to specify the size)
 - `String / std::string`
 - `Print / std::ostream`

The previous chapters cover most of these functions.

5.5 Inside JsonObject

Implementation

JsonObject is simply a collection of JsonPairs, it is implemented as a linked-list. A JsonPair is a structure that packs a key and a JsonVariant together. Each node of the linked-list is allocated in the JsonBuffer. JsonObject holds a pointer to its JsonBuffer, allowing it to allocate more nodes when needed.

Here is a simplified definition:

```
struct JsonPair {
    const char* key;
    JsonVariant value;
};

struct JsonObjectNode {
    JsonPair pair;
    JsonObjectNode *next;
};

struct JsonObject {
    JsonObjectNode *head;
    JsonBuffer *buffer;
};
```

Creating a JsonObject

JsonObject's constructor is private, so it's not possible to simply declare a JsonObject and use it. Instead, you must call `JsonBuffer::createObject()` which returns a reference to a JsonObject:

```
JsonObject obj;
StaticJsonBuffer<200> jb;
JsonObject& obj = jb.createObject();
```

See chapter [“Serialize with ArduinoJson”](#) for a tutorial.

Parsing a JsonObject

To parse a JsonObject, you need to use the JsonBuffer too:

```
StaticJsonBuffer<200> jb;  
JsonObject& arr = jb.parseObject(input);
```

See chapter [“Deserialize with ArduinoJson”](#) for a tutorial.

Invalid

When `JsonBuffer::createObject()` and `JsonBuffer::parseObject()` fail, they return a reference to a special instance of `JsonObject`. You can get this reference by calling `JsonObject::invalid()`.

This special instance implements the null-object pattern. The null-object has no behavior, so all its methods do nothing. It simplifies the program because you don’t need to check that the return value is not null. See chapter [“Deserialize with ArduinoJson”](#) for an example.

You’ll probably never call `JsonObject::invalid()` in your program, but `ArduinoJson` uses this function every time it cannot return a reference to a `JsonObject`.

Copying a JsonObject

`JsonObject`’s copy-constructor and copy-assignment are private, so it’s not possible to copy a `JsonObject`. If you need to pass a `JsonObject` to a function, you must use a reference.

I’ve never seen the case, but if you want to make a complete copy of a `JsonObject`, you need to iterate through all the key-value pairs of the first and add them to the seconds.

JsonObject as a generic container

I’ve seen several programs using a `JsonObject` as a generic dictionary/map. It is a bad idea for three reasons:

1. The versatility of `JsonVariant` causes a significant overhead in speed and memory usage, compared to a custom data structure.
2. `JsonObject::remove()` can remove a node from the list but doesn’t release the memory in the `JsonBuffer`, causing a memory leak.
3. The lookup in a `JsonObject` has a $O(n)$ complexity, so the performance quickly drops when the number of elements increases.

In the [case studies](#), we’ll use `ArduinoJson` to store the configuration of a program; as you’ll see, there is no need to use a global `JsonObject` as a generic container.

Methods

A detailed list of methods and overloads is available in the [“API Reference” on arduinojson.org](https://arduinojson.org/api-reference/). Here is a summary:

- `begin()` and `end()` return STL-style iterators.
- `createNestedArray(key)` associates a new `JsonArray` with the specified key, and returns a reference.
- `createNestedObject(key)` associates a new `JsonObject` with the specified key, and returns a reference.
- `get<T>(key)` converts the `JsonVariant` associated with the specified key to the type `T`, and returns a default value (like `0`, `0.0` or `nullptr`) if the type is incompatible.
- `invalid()` returns a reference to the null-object.
- `is<T>(key)` tells if the `JsonVariant` at specified key has the type `T`.
- `measureLength()` computes the size of the minified JSON document.
- `measurePrettyLength()` computes the size of the prettified JSON document.
- `operator[] (key)` provides an alternative syntax for `set()` and `get()`.
- `prettyPrintTo(destination)` produces a prettified JSON document, and returns the number of bytes written.
- `printTo(destination)` produces a minified JSON document, and returns the number of bytes written.
- `remove(key)` removes the `JsonVariant` at specified key but doesn't release the memory in the `JsonBuffer`.
- `set(key, value)` replaces the `JsonVariant` at specified key.
- `size()` returns the number of key-value pairs in the `JsonObject`.
- `success()` tells if the `JsonObject` is valid (i.e. not the null-object).

In the methods above:

- key's type can be:
 - `char*` / `const char*` / `char[]`
 - `String` / `std::string`
 - `const __FlashStringHelper*`
- value's type or `T` can be:
 - `bool`
 - `char` / `short` / `int` / `long` / `long long` (all signed or unsigned)
 - `float` / `double`
 - `String` / `std::string`
 - `JsonArray` / `JsonObject`
 - `char*` / `const char*` / `char[]`

- const __FlashStringHelper*
- RawJson
- JsonVariant
- destination's type can be:
 - char[]
 - char* (plus an integer to specify the size)
 - String / std::string
 - Print / std::ostream

The previous chapters cover all these functions.

Remark on operator[]

The return type of `JsonObject::operator[]()` is *conceptually* a `JsonVariant&`.

In practice, it's not a `JsonVariant&` because reading a value from the array would force the library to create the association.

For example:

```
if (obj["hello"] == 42) {  
  
}
```

In the code above, if `obj["hello"]` returned a `JsonVariant&`, it would need to create a new `JsonPair`, set the key to "hello" and return a reference to the variant. In the C++ Standard Library, `std::map` works like that.

ArduinoJson is different: reading a value doesn't modify the `JsonObject`. This feature is implemented by returning another type from `operator[]`. This type is `JsonObjectSubscript`, you don't have to use it in your program, but you may occasionally see its name in error messages.

5.6 Inside JsonVariant

Implementation

JsonVariant is just a union of the types allowed in a JSON document, alongside with an enum to select the type.

In C++, a union is a structure where every member overlaps, they all use the same bytes. The size of a union is, therefore, the size of its biggest member. A program needs to know which member of the union is valid before using it. JsonVariant uses an enum to remember which member of the union is the right one.

Here is a simplified definition of JsonVariant:

```
union JsonVariantValue {
    const char* stringValue;
    double doubleValue;
    long longValue;
    bool boolValue;
    JsonArray* arrayValue;
    JsonObject* objectValue;
};

enum JsonVariantType {
    JSON_VARIANT_UNDEFINED,
    JSON_VARIANT_STRING,
    JSON_VARIANT_DOUBLE,
    JSON_VARIANT_LONG,
    JSON_VARIANT_BOOL,
    JSON_VARIANT_ARRAY,
    JSON_VARIANT_OBJECT
};

struct JsonVariant {
    JsonVariantValue value;
    JsonVariantType type;
};
```

As you see, a JsonVariant stores pointers to strings; that's why ArduinoJson duplicates the strings into the JsonBuffer to make sure that the pointer is valid. It also stores pointers to arrays and objects; so, when you add an array or an object as a member of another one, it is added by reference, not by copy.

Undefined

If no value is provided to the constructor, a `JsonVariant` is “undefined”:

1. `is<T>()` returns `false` for all `Ts`,
2. `as<T>()` returns the default value for all `Ts`,
3. `success()` returns `false`.

Remark that `undefined` is not a valid token in a JSON document. In the current implementation of `ArduinoJson`, an undefined `JsonVariant` doesn't produce any output.

The unsigned long trick

In the simplified definition above, I said that `ArduinoJson` stores integral values as `long`, but it is more complicated than that.

Consider this example:

```
JsonArray& arr = jb.createArray();  
arr.add(4294967295UL); // biggest unsigned long
```

You naturally expects `arr` to be serialized as:

```
[4294967295]
```

But, if it was stored in a `long`, it would be serialized as:

```
[-1]
```

Indeed, there is an overflow: `(long)4294967295UL` becomes `-1`.

To work around this problem, `ArduinoJson` uses different enum values for positive and negative values, and everything works as expected.

ArduinoJson's configuration

`ArduinoJson` has a few compile-time settings that affect the definition of `JsonVariant`:

ARDUINOJSON_USE_DOUBLE

ARDUINOJSON_USE_DOUBLE determines whether to use `double` or `float`.

When set to 1, it uses `double`. Floating points values are stored with a better precision (up to 9 digits), but `JsonVariant` is much bigger. This mode is the default when the target is a computer.

When set to 0, it uses `float`. The precision is lower (up to 6 digits), but `JsonVariant` is smaller. This mode is the default when the target is an embedded platform.

ARDUINOJSON_USE_LONG_LONG

ARDUINOJSON_USE_LONG_LONG determines whether to use `long long` or `long`.

When set to 1, it uses `long long`. Integral values are stored in a 64-bit integer, but the `JsonVariant` is bigger. This mode is the default when the target is a computer.

When set to 0, it uses `long`. Integral values are stored in a 32-bit integer only, but `JsonVariant` is smaller. This mode is the default when the target is an embedded platform.



The ArduinoJson Assistant assumes the defaults

If you change the default value, it affects the result of `JSON_ARRAY_SIZE()` and `JSON_OBJECT_SIZE()`, therefore the ArduinoJson Assistant will compute wrong results.



Don't mix and match!

A program composed of multiple compilation units (i.e. several `.cpp` files) must have the same configuration in all of them. If ArduinoJson has a different configuration in a compilation unit, the program would compile without any error, but undefined behavior would pop up at run-time.

Iterating through a `JsonVariant`

We saw in a previous chapter how to enumerate [all values in a `JsonArray`](#) and [all key-value pairs of a `JsonObject`](#), but doing the same with a `JsonVariant` is a bit more complicated.

As a `JsonVariant` can be a `JsonArray` or a `JsonObject`, we need to help the compiler and tell it which type we expect. We can do that with an implicit cast, or with a call to `as<T>()`.

Let's see two concrete examples of a nested object and a nested array.

An object in an array

Here is an example of an object nested in an array:

```
[
  {
    "hello": "world"
  }
]
```

You can enumerate the key-value pairs of the object, by casting the `JsonVariant` to a `JsonObject`:

```
JsonObject& arr = jb.parseArray(input);

for(JsonPair& kvp : arr[0].as<JsonObject>()) {
  const char* key = kvp.key;    // "hello"
  JsonVariant value = kvp.value; // "world"
}
```

An array in an object

Here is an example of an array nested in an object:

```
{
  "results": [
    1,
    2
  ]
}
```

You can enumerate the element of the array, by casting the `JsonVariant` to a `JsonArray`:

```
JsonObject& obj = jb.parseObject(input);

for(int result : obj["results"].as<JsonArray>()) {
  // result = 1, then 2...
}
```



What happens if the type is incompatible?

It is safe to do the conversion to `JsonArray` or `JsonObject`, even if the `JsonVariant` doesn't contain the expected type, there will simply be no iteration of the loop.

For example, if you call `as<JsonArray>()` on a `JsonVariant` that contains a `JsonObject`, the function return `JsonArray::invalid()`. `JsonArray::invalid()` is the null-object, which behaves like an empty array, so no iteration is made.

The or operator

As we saw in the chapter “[Deserialize with ArduinoJson](#)”, you can use the “or” operator (`|`) to provide a default value in case the value is missing or is incompatible. Here is an example:

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

This operator doesn’t use the implicit cast. Instead, it looks at the type of the default value provided to the right. If the variant is compatible, it returns the value; otherwise, it returns the default value. Here is a simplified implementation of this operator:

```
template<typename T>
T operator|(const JsonVariant& variant, T defaultValue) {
    return variant.is<T>() ? variant.as<T>() : defaultValue;
}
```

But, there is one exception. If the `JsonVariant` contains `null`, even if `is<char*>()` returns `true`, the operator returns the default value. This different behavior has been added to stop the propagation of `null`, it is leveraged by the sample below:

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

Indeed, `strncpy()` doesn’t allow `nullptr` as a second parameter; using the “or” operator protects us from this case.

Methods

A detailed list of methods and overloads is available in the “[API Reference](#)” on arduinojson.org. Here is a summary:

- `as<T>()` converts the variant to the type `T`, returns a default value (like `0`, `0.0` or `nullptr`) if the type is incompatible.
- `is<T>()` tells if the variant at specified key has the type `T`.
- `operator T()` is a synonym of `as<T>()`
- `operator [] (index)` mimics a `JsonArray`, allowing expressions like `obj["results"][0]`.
- `operator [] (key)` mimics a `JsonObject`, allowing expressions like `obj["location"]["latitude"]`.
- `operator==(value)` tells if the variant is equal to the specified value.

- `operator<=(value)` tells if the variant is lower or equal to the specified value.
- `operator<(value)` tells if the variant is lower to the specified value.
- `operator>=(value)` tells if the variant is higher or equal to the specified value.
- `operator>(value)` tells if the variant is higher to the specified value.
- `operator|(defaultValue)` converts the variant to the specified type, returns `defaultValue` if the type is incompatible.
- `success()` tells if the variant has a value.

In the methods above, `T` and `value` can be:

- `bool`
- `signed / unsigned char / short / int / long / long long`
- `float / double`
- `String / std::string`
- `JsonArray / JsonObject`
- `char* / const char* / char[]`
- `const __FlashStringHelper*`
- `RawJson`
- `JsonVariant`

5.7 Inside the parser

Invoke the parser

The parser is the piece of code that performs the text analysis. It is used during the deserialization process to convert the JSON document into the `JsonArrays`, `JsonObject`s and `JsonVariants`. With `ArduinoJson`, a program invokes the parser through a `JsonBuffer`.

`JsonBuffer` has three functions to invoke the parser:

1. `parseArray()` which returns a `JsonArray&`
2. `parseObject()` which returns a `JsonObject&`
3. `parse()` which returns a `JsonVariant`

We already covered the two firsts in the chapter [“Deserialize with ArduinoJson”](#). The third function is similar, except that it works with array and objects; use it when you don’t know what the content of the input is.

Here is an example:

```
// Allocate the JsonBuffer
StaticJsonBuffer<200> jb;

// Parse the JSON document in input
JsonVariant v = jb.parse(input);

// Is it an object?
if (v.is<JsonObject>()) {
    // Yes! Convert it to an object.
    JsonObject &obj = v;
    // ...
}

// Is it an array?
if (v.is<JsonArr>()) {
    // Yes! Convert it to an object.
    JsonArr &arr = v;
    // ...
}
```

You can a complete program using this feature in [the case studies](#).

The JSON document above contains 15 opening brackets. When the parser reads this document, it enters in 15 levels of recursions. If we suppose that each recursion adds 8 bytes to the stack (the size of the local variables, arguments and return address), then this document adds up to 120 bytes to the stack. Imagine what we would get with more nesting levels...

This overflow is dangerous because a malicious user could send a specially crafted JSON document that makes your program crash. In the best case scenario, it just causes a Denial of Service (DoS); but in the worse case, the attacker may be able to alter the variables of your program.

As it is a security risk, ArduinoJson puts a limit on the number of nesting levels allowed. This limit is 10 on an embedded platform and 50 on a computer. You can temporarily change the limit by passing the new value to the second argument of `parseArray()`, `parseObject()` or `parse()`. Alternatively, you can change the limit for the whole program by changing `ARDUINOJSON_DEFAULT_NESTING_LIMIT`.

The program below raises the nesting limit, so it can read the JSON document above:

```
const int size = 15*JSON_ARRAY_SIZE(1);
StaticJsonBuffer<size> jb;

JsonArray& arr = jb.parseArray(input, 15);

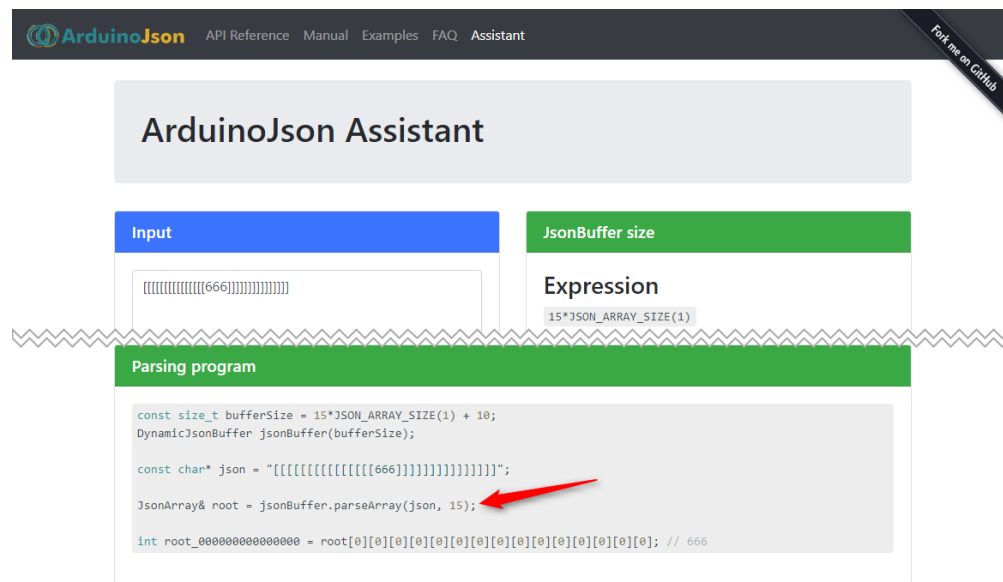
int value = arr[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]; // 666
```



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 sample program changes the nesting limit when necessary.



Quotes

The JSON specification states that strings are delimited by double quotes ("), but JavaScript allows single quotes (') too. In fact, JavaScript even allows object keys without quotes when there is no ambiguity.

Here is an example that is valid JavaScript, but not in JSON:

```
{
  hello: 'world'
}
```

This example works because the key `hello` only contains alphabetic characters, but as soon as the key includes punctuations or spaces, we must use quotes.

Like JavaScript, ArduinoJson accepts:

- double quotes (")
- single quotes (')
- no quote

Escape sequences

The JSON specification defines a list of escape sequences that can be inserted in a string to place a special character.

For example:

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

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

ArduinoJson handles the following escape sequences:

Escape sequence	Meaning
<code>\"</code>	Quote (<code>"</code>)
<code>\/</code>	Forward slash
<code>\\</code>	Backslash (<code>\</code>)
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tabulation

When parsing a document, ArduinoJson replaces each escape sequence with its matching ASCII character. Similarly, when serializing, it replaces each character from the list with the matching escape sequence.

However, ArduinoJson doesn't support the following escape sequences:

Escape sequence	Meaning
<code>\/</code>	Forward slash (<code>/</code>)
<code>\uXXXX</code>	UTF-16 surrogate pair

The first is useless, as you can write a slash (`/`) in a string without escaping it. The second is barely used in practice, because almost everyone encodes JSON document in UTF-8, in which case escape sequences are not required.

As converting UTF-16 escape sequence to UTF-8 is non-trivial, many JSON parsers, including ArduinoJson, don't support it. This feature would not be worth the extra bytes in the executable because nobody uses them in practice.

Comments

The JSON specification does not allow writing comments in a document, but JavaScript does.

Here is an example that is valid JavaScript, but not in JSON:

```
{  
  /* a block comment */  
  "hello": "world" // a trailing comment  
}
```

ArduinoJson can read a document containing comments; it simply ignores them. However, ArduinoJson is not able to write comments in a document. As a consequence, if you deserialize then serialize a document, you lose all the comments.

Stream

A useful property of the ArduinoJson parser is that it tries to read as few bytes as possible. This behavior is helpful when parsing from a stream because it will stop 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. We'll see two examples in the [case studies](#) that leverage this feature to reduce the memory consumption.

This behavior also allows ArduinoJson to read inputs that are not null-terminated. However, I don't recommend to do that because it's very risky.

5.8 Inside the serializer

Invoke the serializer

The serializer is the piece of code that converts `JsonArrays`, `JsonObject`s and `JsonVariants` to a JSON document. The serializer is a hidden part of `ArduinoJson`; you don't directly interact with it. Instead, you call a method on the object you want to serialize.

The following methods of `JsonArray`, `JsonObject` and `JsonVariant` invoke the serializer:

1. `printTo()` which produces a minified JSON document,
2. `prettyPrintTo()` which produces a prettified JSON document.

Both methods take one argument, the destination of the document, and return the number of bytes written.

The type of the destination can be:

- `char[]`
- `char*`, in that case, you must specify the size
- `Print` or `std::ostream`
- `String` or `std::string`



Why the name `printTo()`?

This name is a legacy from a previous version of `ArduinoJson`. In that version `JsonArray` and `JsonObject` derived from the abstract class `Printable` which is part of `Arduino`. It was convenient because it allowed writing the following:

```
Serial.print(obj);
```

Unfortunately, I had to remove this inheritance to reduce the size of the objects.

Measure the length

They are two other methods which invoke the serializer:

1. `measureLength()` which computes the size of the minified JSON document,
2. `measurePrettyLength()` which computes the size of the prettified JSON document.

In fact, these two methods are just thin wrappers on top of `printTo()` and `prettyPrintTo()`. They pass a dummy implementation of `Print` that does nothing but count the number of bytes written.



Performance

`measureLength()` is a costly operation because it involves doing the complete serialization, use it only if you must.

Escape sequences

Regarding string escape sequence, the serializer supports [the same features](#) and have [the same limitations](#) as the parser. It converts the special ASCII characters into the matching sequences, reversing the work of the parser.

Float to string

Since version 5.10, ArduinoJson has a unique float-to-string conversion specially optimized for:

1. low memory consumption,
2. small code size,
3. speed.

But it has some limitations:

1. it prints only nine digits after the decimal point,
2. it doesn't allow to change the number of digits,
3. it doesn't allow to change the exponentiation threshold.

I explained how this algorithm works in the article [“Lightweight float to string conversion”](#) in my blog. I invite you to check if you're interested.

After introducing this algorithm, I received a few complains because the result has too many digits after the decimal point. It's not a problem with the accuracy of the conversion, but these users want a value rounded to 2 decimal places, as it was in the previous version.

As stated above, we cannot specify the number of digits, as we used to. But there is a very simple workaround; we just need to perform the rounding in our program:

```
// Rounds a number to 2 decimal places  
double round2(double value) {  
    return (int)(value * 100 + 0.5) / 100.0;  
}
```

```
obj["pi"] = round2(3.14159); // 3.14
```

We could also use `RawJson()` with a `String` to get the same result:

```
obj["pi"] = RawJson(String(3.14159)); // 3.14
```

But this version uses dynamic memory allocation, which is bad.

5.9 Miscellaneous

The ArduinoJson namespace

Every single piece of the library is defined in the `ArduinoJson` namespace. However, since using a namespace is not a common practice for Arduino libraries, the last line of `ArduinoJson.h` is a `using namespace` statement. However, there is another header, `ArduinoJson.hpp`, that doesn't have this line.

If you want to keep `ArduinoJson` in its namespace, simply include `ArduinoJson.hpp` instead of `ArduinoJson.h`.

```
#include <ArduinoJson.h>
#include <ArduinoJson.hpp>

void setup() {
  StaticJsonBuffer<200> jb;
  ArduinoJson::StaticJsonBuffer<200> jb;
  // ...
}
```

`JsonBuffer::clear()`

`JsonBuffer` has a `clear()` method that resets it to its initial state.

You may wonder why I'm waiting for the end of the chapter to talk about this method. The reason is that I don't think it should be in the library, it's an error in the history of `ArduinoJson`.

I said countless times that one should not reuse a `JsonBuffer`, but instead destroy and create a new one. If you follow the patterns presented in this book, you should never feel the need for a `clear()` method. In the [case studies](#), we'll see that reusing a `JsonBuffer` is not required.



Code smell

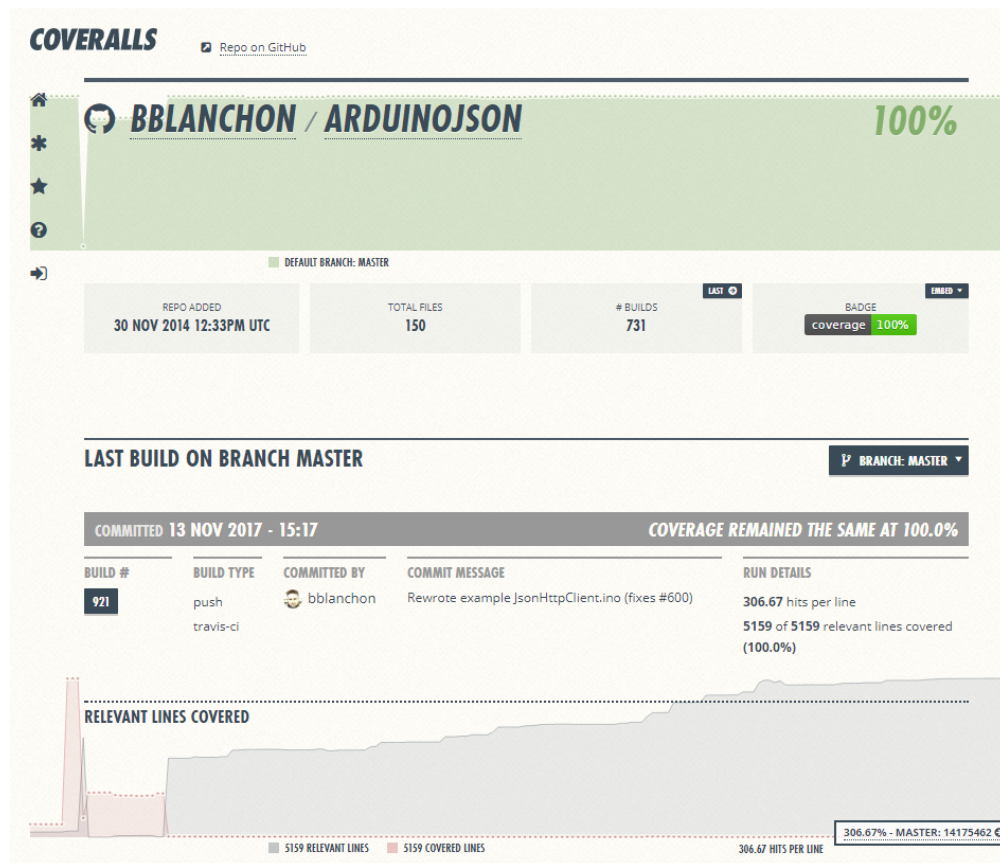
If you see this function in a program, it means that `ArduinoJson` is not used properly, for example:

- The `JsonBuffer` is global, causing a huge waste of RAM.
- A `JsonObject` is used as a generic container (to store the configuration for example), wasting RAM and CPU cycles.
- The `JsonBuffer` is defined outside of a loop instead of being defined inside the loop.

Code coverage

The development of ArduinoJson followed the discipline of Test Driven Development (TDD), which imposes that one must write the tests before the production code. This technique allowed to add many features to the library with relatively few bugs.


ArduinoJson is probably the only Arduino library to have a code coverage of 100%, meaning that every single line of code is verified by a unit test. The coverage status is monitored on coveralls.io, and you can find the link on the GitHub page.



Coveralls.io

Fuzzing

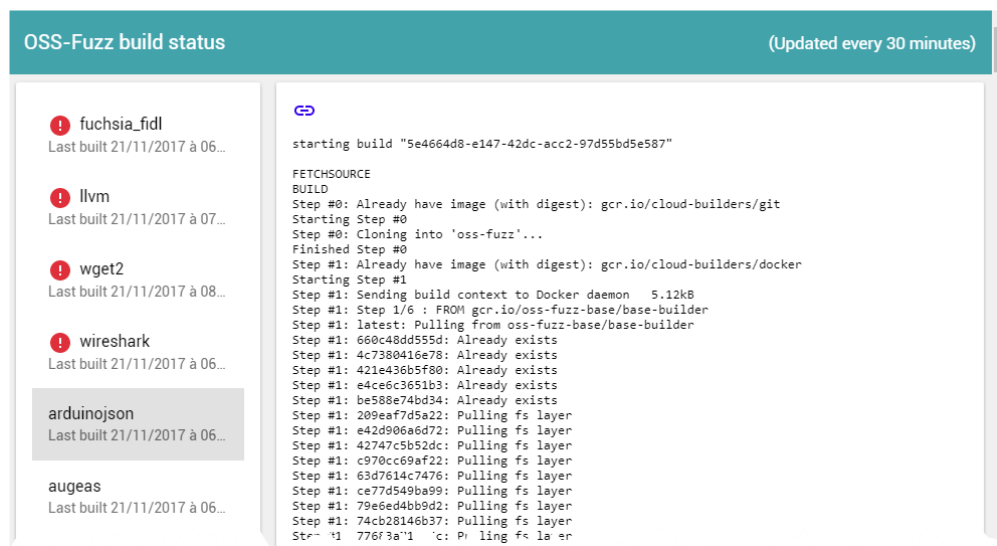
As far as I know, only one vulnerability has been found in ArduinoJson. In 2015, a bug in the parser allowed an attacker to crash the program with a malicious JSON document. This vulnerability was filed under the reference [CVE-2015-4590](https://cve.mitre.org/cve/2015/4590/). This bug was fixed in ArduinoJson 4.5, on the same day it was discovered.



The screenshot shows the CVE-2015-4590 page on the Common Vulnerabilities and Exposures (CVE) website. The page title is "CVE-2015-4590" and it is categorized under "Buffer overflow in ArduinoJson when parsing crafted JSON strings". The description states: "The extractFrom function in Internals/QuotedString.cpp in Arduino JSON before 4.5 allows remote attackers to cause a denial of service (crash) via a JSON string with a \ (backslash) followed by a terminator, as demonstrated by '\\0', which triggers a buffer overflow and over-read." The references section lists several links, including the MLIST entry, the OpenWall CVE list, and the GitHub pull request for the fix.

CVE-2015-4590

A student found this bug with a technique called “fuzzing” which consists in sending random input into the parser until the program crashes. After this incident, I added ArduinoJson to OSS-Fuzz, a project led by Google, that performs continuous fuzzing on open-source projects.



The screenshot shows the OSS-Fuzz build status page. On the left, there is a list of projects with their build status and last built time. The projects listed are fuchsia_fidl, llvm, wget2, wireshark, arduinojson, and augeas. The arduinojson project is highlighted. On the right, there is a detailed log of the build process for arduinojson, showing the steps from fetching the source to building the project.

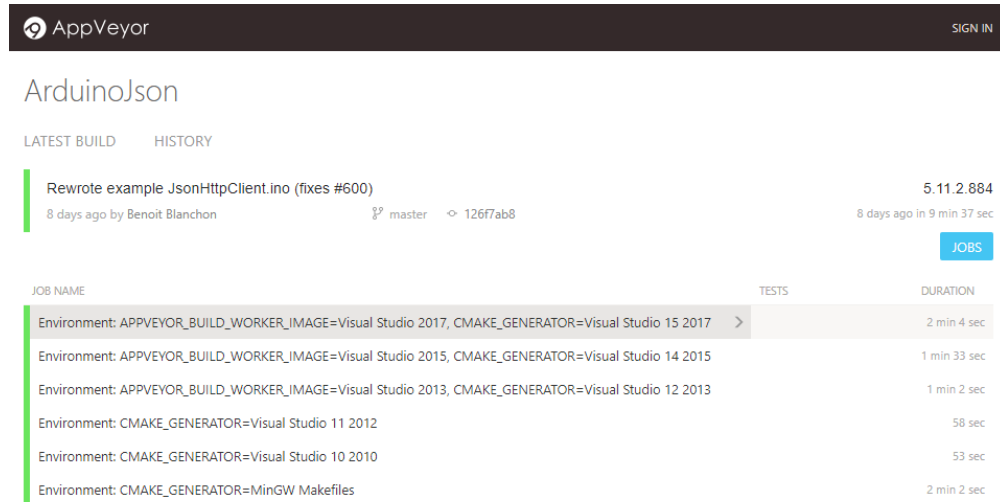
OSS-Fuzz status

Portability

ArduinoJson is very portable, meaning that the code can be compiled for a wide range of targets. Indeed, the library has very few dependency, because many parts, like the float-to-string conversion, are implemented from scratch. In particular, ArduinoJson doesn't depend on Arduino; so you can use it in any C++ project.

I've been able to compile ArduinoJson with all compilers that I had in hand, with one notable exception: Embarcadero C++ Builder.

Continuous Integration (CI) is a technique that consists in automating a build and test of the library each time the source code changes. ArduinoJson uses two web services for that: AppVeyor and Travis (see screen captures below).



JOB NAME	TESTS	DURATION
Environment: APPVEYOR_BUILD_WORKER_IMAGE=Visual Studio 2017, CMAKE_GENERATOR=Visual Studio 15 2017	>	2 min 4 sec
Environment: APPVEYOR_BUILD_WORKER_IMAGE=Visual Studio 2015, CMAKE_GENERATOR=Visual Studio 14 2015		1 min 33 sec
Environment: APPVEYOR_BUILD_WORKER_IMAGE=Visual Studio 2013, CMAKE_GENERATOR=Visual Studio 12 2013		1 min 2 sec
Environment: CMAKE_GENERATOR=Visual Studio 11 2012		58 sec
Environment: CMAKE_GENERATOR=Visual Studio 10 2010		53 sec
Environment: CMAKE_GENERATOR=MinGW Makefiles		2 min 2 sec

Continuous Integration on AppVeyor

Travis CI About Us Blog Status Help Benoît Blanchon

bblanchon / ArduinoJson build passing

Current Branches Build History Pull Requests More options

✓ **master** Rewrote example JsonHttpClient.ino (fixes #600) → #921 passed Restart build

→ Commit 126f7ab
 ↕ Compare 2eab5b0...126f7ab
 ↗ Branch master
 Benoît Blanchon authored and committed

Ran for 31 min 55 sec
 Total time 1 hr 3 min 10 sec
 8 days ago

Build Jobs

Job	Compiler	Script	Duration
✓ # 921.1	</> Compiler: gcc C++	SCRIPT=cmake GCC=4.4	2 min 5 sec
✓ # 921.2	</> Compiler: gcc C++	SCRIPT=cmake GCC=4.6	2 min 11 sec
✓ # 921.3	</> Compiler: gcc C++	SCRIPT=cmake GCC=4.7	2 min 12 sec
✓ # 921.4	</> Compiler: gcc C++	SCRIPT=cmake GCC=4.8 SANITIZE=addr...	3 min
✓ # 921.5	</> Compiler: gcc C++	SCRIPT=cmake GCC=4.9 SANITIZE=leak	3 min 21 sec
✓ # 921.6	</> Compiler: gcc C++	SCRIPT=cmake GCC=5 SANITIZE=undefi...	4 min 17 sec
✓ # 921.7	</> Compiler: gcc C++	SCRIPT=cmake GCC=6	3 min 47 sec
✓ # 921.8	</> Compiler: gcc C++	SCRIPT=cmake GCC=7	5 min 43 sec
✓ # 921.9	</> Compiler: clang C++	SCRIPT=cmake	1 min 42 sec
✓ # 921.10	</> Compiler: clang C++	SCRIPT=cmake CLANG=3.5 SANITIZE=a...	3 min 17 sec

Continuous Integration on Travis

Here are the compilers used by the CI:

Compiler	Versions
Visual Studio	2010, 2012, 2013, 2015, 2017
GCC	4.4, 4.6, 4.7, 6, 7
Clang	3.5, 3.6, 3.7, 3.8

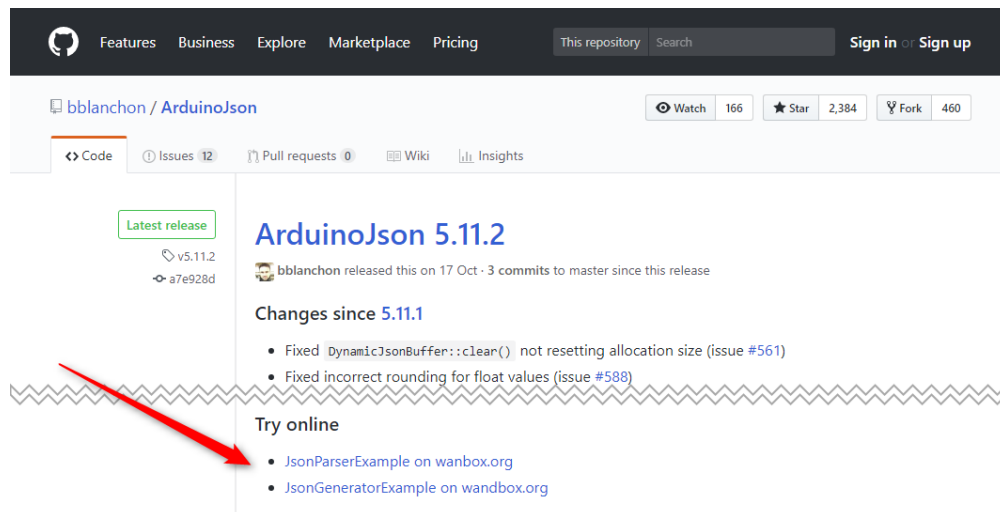
As this table attests, the portability of ArduinoJson is considered seriously. Needless to say that the CI forbids any error and any warnings.

Online compiler

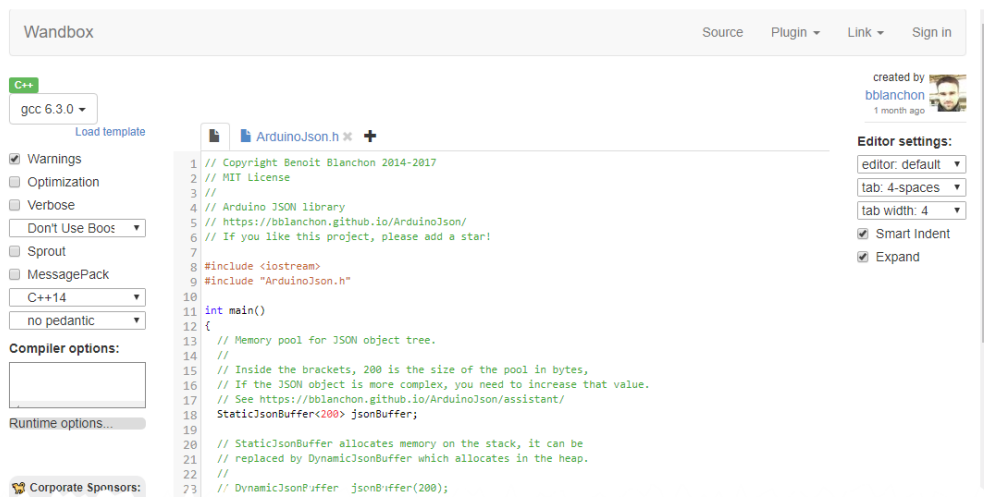
ArduinoJson is self-contained (it has no dependency) and fits in a single header file, which makes it a perfect candidate for online compilers. You just need to copy the content of the single header version of ArduinoJson (see [Introduction](#)) and paste it into the online compiler.

Every release of ArduinoJson is accompanied by links to examples on wandbox.org, so you can try

the library online. That is very handy when you experiment with the library; for example when you want to demonstrate some issue. I often use wandbox.org to answer questions on GitHub.



Links to wandbox.org on release page



JsonParserExample on wandbox.org

License

ArduinoJson is released under the terms of the MIT license, which is very permissive.

- You can use ArduinoJson in closed-source projects.
- You can use ArduinoJson in commercial applications without redistributing royalties.
- You can modify ArduinoJson without publishing the source.

Your only obligation is to provide a copy of the license, including the name of the author, with every copy of your software. But I promise I won't sue if you forget :-)

6. Troubleshooting



The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

– Brian Kernighan, Unix for Beginners

6.1 Program crashes

Undefined Behaviors

Like any C++ code, your program is going to crash if it contains a bug.

Some bugs are very simple to find and understand, like dereferencing a null pointer; but, most of the time, it's very difficult because the program may work for a while, and then fails for an unknown reason.

In the C++ jargon, we call that an “Undefined Behavior” or “UB”. The C++ specification and the C++ Standard Library contains 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 null
if(s == nullptr) {} // <- UB!
```

Fortunately, the `String` class of Arduino is different, but it still has some vulnerabilities.

A bug in ArduinoJson?

If your program crashes, especially if it fails when calling a function of `ArduinoJson`, don't blame the library too quickly.

By design, `ArduinoJson` never makes the program crash. As we saw, it implements the [null-object design pattern](#), which protects your program from dereferencing a null pointer when a memory allocation fails or when a value is missing.

However, if you feed `ArduinoJson` with dangling pointers, it may cause the program to crash. The same thing happens if the program uses objects that have been destroyed.



It's not a bug in the library

Many `ArduinoJson` users reported crashes that they attributed to the library, but a tiny fraction was actually due to a bug in the library.

The bug is always in the program, never in the library.

Null string

Here is one common cause of crash: dereferencing a null string returned by ArduinoJson. For example, it can happen with the code below:

```
// Compare the password with the string "secret"
if (strcmp(obj["pwd"], "secret") != 0) { // <- UB if obj["pwd"] is null

}
```

strcmp() is a function of the C Standard Library that compares two strings and returns 0 when they are equal. Unfortunately, the specification states that the behavior is undefined if one of the two strings is null.

Since the behavior is undefined, the code above could work on one platform but crash on another. In fact, this example works on an ATmega328 but crashes on ESP8266.

By the way, you can fix the program above by replacing strcmp() by the equal operator (==) of JsonVariant:

```
// Compare the password with the string "secret"
if (obj["password"] == "secret") {

}
```

Use after free

The term “use-after-free” is used by security professionals to designate a frequent error in C and C++ programs. It refers to a heap variable being used after being freed. A bug like this is very likely to corrupt the memory of the heap, and ultimately to cause a crash, but it can also have no incidence. It can also expose a security risk if an attacker manages to exploit the way the memory is corrupted.

Here is an obvious instance of use-after-free:

```
// Allocate memory in the heap
int* i = (int*)malloc(sizeof(int));

// Release the memory
free(i);

// Use-after-free!
*i = 666;
```

The program dereferences the pointer after freeing the memory. It's easy to see the bug here because `free()` is called explicitly. But it can be more subtle in a C++ program, where the destructor releases the memory.

Here is an example where the bug is more difficult to see:

```
// Returns the name of the digital pin: D0, D1...
const char* pinName(int id) {
    // Construct a local variable with the name
    String name = String("D") + id;

    // Return in the form of a const char*
    return name.c_str();

    // The local variable "name" is destructed as soon as the function exits
}

// 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 JsonBuffer
    DynamicJsonBuffer jb;

    // Parse the array with the names of the pins
    JsonArray &pins = jb.parseArray(json);

    // Return the name of the pin
    return pins[id];

    // The local variable "jb" is destructed as soon as the function exits
}

// Use-after-free!
Serial.println(pinName(0));
```

And a last one, just to show that a function is not necessary:

```
// Create an empty JSON object
JsonObject& obj = jb.createObject();

// Set the name of the pin
obj["pin"] = (String("D") + pinId).c_str();
// The temporary String is destructed after the semicolon (;)

// Use-after-free
obj.printTo(Serial);
```

Return of stack variable address

The example above used a `DynamicJsonBuffer` because the topic was use-after-free, so it had to be in the heap. It's possible to do a similar mistake using a variable on the stack, but this vulnerability is called a "return-of-stack-variable-address". Just replace `DynamicJsonBuffer` by `StaticJsonBuffer`, and you have one. As with use-after-free, the program may or may not work, and may also be vulnerable to exploits.

Here is an obvious example of return-of-stack-variable-address:

```
// Returns the name of the digital pin: D0, D1...
const char* pinName(int id) {
    // Declare a local string
    char name[4];

    // Format the name of the pin
    sprintf(name, "D%d", id);

    // Return the name of the pin
    return name;

    // The local variable "name" is destructed as soon as the function exits
}

// Use destroyed variable "name"!
Serial.println(pinName(0));
```

Here is another involving `ArduinoJson`:

```

// Returns an object with the configuration
JsonObject& readConfig() {
    // Allocate a JsonBuffer
    StaticJsonBuffer<256> jb;

    // Parse the configuration file
    return jb.parseObject(config);

    // The local variable "jb" is destructed as soon as the function exits
    // Remember that the JsonObject was inside the JsonBuffer, so it's gone too.
};

// Use destroyed variable "jb"!
Serial.println(readConfig()["pins"][0]);

```

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

```

Buffer overflows are well known by security professionals because they are ubiquitous and often allow to modify the behavior of the program. strcpy(), sprintf() and the like, are traditional sources of buffer overflow.

Here is a program using ArduinoJson, that presents a serious risk:

```
// Parse a JSON input
JsonObject& obj = jb.parseObject(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 `obj["ip"]` contains a string longer than 16? A buffer overflow. This bug is very dangerous if the JSON document comes from an untrusted source. An attacker could craft a special JSON document that would change the behavior of the program. If you think this is far-fetched, I invite you to learn more about software security from dedicated books and websites.

By the way, you can fix the code above by using `strncpy()` instead of `strcpy()`. `strncpy()`, as the name suggests, takes an additional parameter which is the size of the destination buffer.

```
// Copy the content into the variable
strncpy(ipAddress, obj["ip"] | "", sizeof(ipAddress));
```

As you see, I also added the or operator (`|`) to avoid the UB if the `obj["ip"]` returns null.



Don't use `strncpy()`

There are two functions in the C Standard Library that are very close: `strncpy()` and `strncpy()`. They both copy strings, they both limit the size, but only `strncpy()` adds the null-terminator.

`strncpy()` is almost as dangerous as `strcpy()`, so make sure to use `strncpy()`.

Stack overflow

A “stack overflow” happens when the stack exceeds its capacity, it can have two different effects:

1. On a platform that limits the stack size (like the ESP8266), an exception is raised.
2. On other platforms (like the ATmega328), the stack and the heap walk on each other's feet.

In the first case, the program is almost guaranteed to crash, which is good. In the second case, the stack and the heap are corrupted, so the program is likely to crash and to be vulnerable to exploits.

With `ArduinoJson`, it happens when you use a `StaticJsonBuffer` that is too big. As a general rule, limit the size of a `StaticJsonBuffer` to half the maximum, so that there is plenty of room for other variables and the call stack (function arguments and return addresses).

By the way, if none of this makes sense, make sure you read the [the C++ course](#) at the beginning of the book



Unpredictable program

You just changed one line of code, and suddenly the program behaves unpredictably? It looks like the processor is not executing the code you wrote?

This is the sign of a stack overflow; you need to reduce the number and the size of variables in the stack.

How to detect 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. Valgrind is a tool that does precisely that.
2. You can build the executable in with a flag that enables instrumentation of the code. Clang and GCC offer `-fsanitize` for that.

But 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++ work and recognize the bugs in the code. These bugs manifest in many ways, so it's impossible to dress an exhaustive list. However, there are risky practices that are likely to cause troubles:

- functions returning a pointer (like `String::c_str()` or `JsonVariant::as<char*>()`)
- functions returning a reference (like `createObject()` or `parseObject()`)
- a pointer living longer than its target
- manual memory management with `malloc()/free()` or `new/delete`

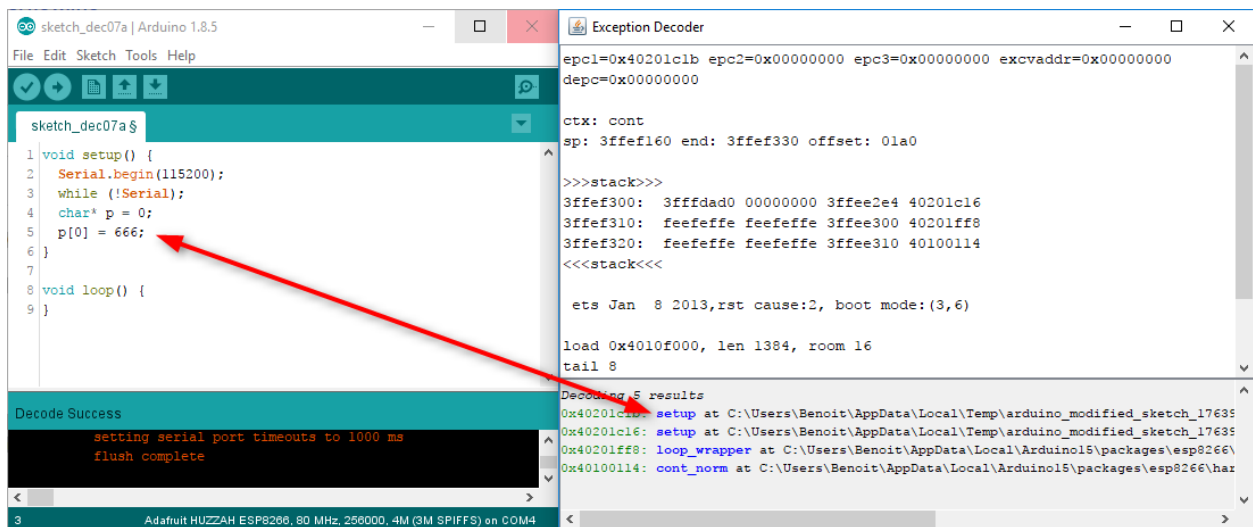
These risky practices should raise a red flag when you review the code, as they are likely to cause trouble. Unfortunately, you cannot eliminate all of them, because there are many legitimate usages too.



Make sense of Exception in ESP8266 and ESP32

When an exception occurs in an ESP8266 or an ESP32, it's logged to the serial port. The log contains the type of the exception, the state of the register and a dump of the stack. There is an excellent tool to extract the call stack, which is very helpful to debug the program.

Visit: github.com/me-no-dev/EspExceptionDecoder



ESP Exception Decoder

6.2 Deserialization issues

A lack of information

In the chapter “[Deserialize with ArduinoJson](#)”, we saw that you can call `JsonObject::success()` to know if the deserialization succeeded:

```
// Parse the JSON document in input
JsonObject& obj = jb.parseObject(input);

// Did it work?
if (!obj.success()) {
    // FAILED!
}
```

And that’s about it. `ArduinoJson` is not very verbose for that matter; it gives no more information.

I’m guilty of that problem. When I created `ArduinoJson`, I was not interested in the reason for a failure. All I wanted to know is if it worked or not. I didn’t care if the failure was due to a memory limitation or to a problem in the input.

What difference would it make anyway? It’s not like I was going to display an error box or record a fancy error message. I was working on a device with very limited capacities, so the only thing that mattered was to build a small and robust program. Knowing that there was an error was sufficient for the job.

This reasoning makes sense when the program is in production, indeed there is very little you can do when the parsing fails in production, all that matters is that the program continues to work. However, it makes it very hard to learn how to use the library, because you don’t know what the problem is.

Don’t worry. It’s not that difficult to figure out what went wrong. We’ll see in this section that there are very few reasons for the deserialization to fail. I’m very confident that, once the learning process is behind you, you’ll enjoy the simplicity of `ArduinoJson`.

Is input valid?

I know it is obvious, but the most common source of failure is a bad input. I received a lot of message from users who didn’t realize that the JSON document was malformed: either the JSON is incorrect, or it’s preceded by some other stuff.

Here are the most common situations for a bad input:

1. The JSON document was handcrafted and contains errors, especially in the punctuation.

2. The JSON document is in the body of an HTTP request (or response), and the headers have not been skipped.
3. The JSON document is in the body of an HTTP 1.1 request, and Chunked Transfer Encoding is not handled correctly.

We already saw how to deal with HTTP in the chapter [“Deserialization with ArduinoJson”](#), and we’ll see more examples in the [case studies](#).

Is the `JsonBuffer` big enough?

The second most common source of failure is when the user copy-pasted one of the examples and forgot to modify the size of the `JsonBuffer`.

This problem is more likely to happen with `StaticJsonBuffer` because `DynamicJsonBuffer` can allocate more memory when needed, but it’s also possible that the `DynamicJsonBuffer` fails to make this allocation.

The solution is simple: increase the size of the `JsonBuffer`. Use the ArduinoJson Assistant to compute the appropriate size.

Also, remember what we said in [previous section about stack-overflow](#). If your target limits the size of the stack, you must switch from a `StaticJsonBuffer` to a `DynamicJsonBuffer` when it’s bigger than half the limit. For example, on an ESP8266, switch to a `DynamicJsonBuffer` as soon as you need more than 2048 bytes; otherwise, you risk causing an exception.

Is there enough RAM?

The third reason for failure is the lack of memory on the device.

Indeed, classic Arduino boards, like the Arduino UNO, have very little RAM. By design ArduinoJson deserializes the whole JSON document into the RAM, so enough memory must be available. For example, you cannot expect to deserialize the huge response from Weather Underground in one shot on a microcontroller that has only 2KB of RAM.

However, before jumping to the conclusion that there is not enough RAM, you should first check that there is no useless duplication. For example, I often see code like this:

```
// Open file
File file = SPIFFS.open(fileName, "r");

// Get file size
size_t size = file.size();

// Allocate a buffer to hold the complete file
std::unique_ptr<char[]> buf (new char[size]);

// Read the whole file
file.readBytes(buf.get(), size);

// Parse JSON
DynamicJsonBuffer jb;
JsonObject& root = jb.parseObject(buf.get());
```

This code works but can be optimized by avoiding the copy of the input and by setting the size of the `JsonBuffer`. Here is the optimized version:

```
// Allocate a JsonBuffer with the perfect size
const int capacity = /* see arduinojson.org/assistant/ */
DynamicJsonBuffer jb(capacity);

// Open file
File file = SPIFFS.open(fileName, "r");

// Parse JSON
JsonObject& root = jb.parseObject(file);
```

Specifying the size of the `JsonBuffer` ensures that the memory is not fragmented (see [previous chapter](#)). Parsing directly from the `File` is more efficient because the parser knows it doesn't have to duplicate spaces, comments, and punctuations.

In the case studies, we'll see a project that use SPIFFS to store its configuration.

If there is still not enough RAM, you have several solutions:

1. change the JSON document if you can (there is more than one forecast service),
2. deserialize in chunks (for example, one day of forecast at a time),
3. upgrade the microcontroller to a bigger one,
4. switch to another library like `json-streaming-parser`.

In the [case studies](#), we'll see two projects that parse a big JSON document in multiple parts.



Zero-copy with a String

If your JSON document is in a `String` and if you know that the `String` is not going to be used after the call to `parseObject()`, you can use the zero-copy mode. To do that, you need to trick `ArduinoJson` into believing that it's a `char*`:

```
JsonObject& root = jb.parseObject(const_cast<char*>(str.c_str()));
```

`const_cast` is a special C++ function that is able to remove the `const`-ness of a type. In this case, it allows converting from `const char*` to `char*`.

How deep is the document?

Remember what we saw in [previous chapter](#): there is a mechanism that rejects documents with too many nesting levels. It is a security feature to protect your program from stack overflows. If your JSON document has more than 10 levels of nesting, you need to pass an extra parameter to `parseObject()` or `parseArray()`.

```
// Increase the allowed nesting levels to 20
JsonObject& obj = jb.parseObject(input, 20);
```

As we said, the `ArduinoJson Assistant` will add this parameter in the sample program.

The first deserialization works?

Your program makes repeated deserializations and only the first (or few firsts) works while the next ones fail? It's because you are reusing the `JsonBuffer` from one deserialization to the other, don't do that.

As we said already, `JsonBuffer` are throw-away memory pools, and they must be short-lived. If you need to perform several deserializations, create a new `JsonBuffer` for each one:

```
// Don't declare the JsonBuffer out of the loop
StaticJsonBuffer<200> jb;

while (true) {
  // Declare the JsonBuffer in the loop
  StaticJsonBuffer<200> jb;

  JsonObject& obj = jb.parseObject(input);
  // ...
}
```



Never declare a global `JsonBuffer`

I said it already, but it's worth repeating: never declare a global `JsonBuffer`! When users want to do that, it's because they have a global `JsonObject` too.

I'm sorry but `ArduinoJson` is not the right tool for that. It is not designed as a generic associative container or property tree. Optimizing this use-case would ruin the performance of `ArduinoJson`, so it's not going to happen.

Follow the examples in the [case studies](#), and you should never feel the need for a global `JsonBuffer` again.

6.3 Serialization issues

Problems in serialization are less frequent than in deserialization because there are fewer reasons to fail.

The JSON document is incomplete

If the generated JSON document misses some part, it's because the `JsonBuffer` is too small.

For example, suppose you want to generate the following:

```
{  
  "firstname": "max",  
  "name": "power"  
}
```

If instead, you get the following output:

```
{  
  "firstname": "max"  
}
```

Then the only thing you need to do is increase the capacity of the `JsonBuffer`.

As usual, use the `ArduinoJson Assistant` to compute the appropriate capacity, which may include additional bytes for the strings that need to be duplicated.

The JSON document contains garbage

If the generated JSON document includes a series of random characters, it's because the `JsonObject` (or the `JsonArray`) 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();
}

// Create an empty object
JsonObject& obj = jb.createObject();

// Add the pin name
obj["pin"] = pinName(0);

// Serialize
obj.printTo(Serial);
```

The author of this program expects the following output:

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

But, she is very likely to get something like that instead:

```
{ "pin": "sÜd4xaÜY9Ė&Q%9;" }
```

The JSON document contains garbage because the `obj["pin"]` store a pointer to a destructed object. Indeed the temporary `String` declared in `pinName()` is destructed as soon as the function exits. By the way, this is an instance of use-after-free, as we saw [earlier in this chapter](#).

There are many ways to fix this program; the simplest is to return a `String` from `pinName()`:

```
// Returns the name of a digital pin: D0, D1...
String pinName(int id) {
    return String("D") + id;
}
```

When we insert a `String` in a `JsonObject`, it makes a copy inside the `JsonBuffer`, so the temporary string can safely be destructed. However, we now need to increase the capacity of the `JsonBuffer`.

Too much duplication

We saw that ArduinoJson stores pointers to strings. If the type of the string is `char*`, `String` or `const __FlashStringHelper*`, ArduinoJson makes a copy in the `JsonBuffer` before saving the pointer. Only strings whose type is `const char*` avoid the duplication.

Duplicating `String` is usually what you expect because they are volatile by nature. However, duplicating a Flash string is probably not what you want, but ArduinoJson needs it. It's probably alright to make one or two copies of a Flash string, but it becomes problematic when the number increases.

Here is an example that fills the `JsonBuffer` with a copy of the same string:

```
// Create an empty array
JsonObject& arr = jb.createObject();

// Add four values to the object
for (int i=0; i<4; i++) {
    // Create an object in the object
    JsonObject& obj = arr.createNestedObject();

    // Set the id of sensor
    obj[F("sensor_id")] = i;
}

// Serialize the object
arr.printTo(Serial);
```

This program produces the expected output:

```
[{"sensor_id":0,"sensor_id":1,"sensor_id":2,"sensor_id":3}]
```

However, the `JsonBuffer` contains four copies of the string `"sensor_id"` which is a waste of memory. This duplication happens because the value is added with a Flash string, but we would have the same problem if `F()` was replaced by `String()`.

The problem can be avoided by not using a Flash string:


```
// Create an empty object
JsonObject& arr = jb.createObject();

// Add four values to the object
for (int i=0; i<4; i++) {
    // Create an object in the object
    JsonObject& obj = arr.createNestedObject();

    // Set the id of sensor
obj["sensor_id"] = i;
    obj["sensor_id"] = i;
}

// Serialize the object
arr.printTo(Serial);
```

We could also duplicate the string explicitly outside of the loop, but the code would be less readable, so it's not worth it.



Don't overuse Flash strings

When used correctly, Flash strings are a good way to save RAM, but when misused, they waste RAM and ruin performance.

Only use Flash string for rarely used long strings, like log messages.

The first serialization succeeds?

If your program repeatedly serializes a JSON document, but only the first (or few firsts) iteration succeed, it is because the `JsonBuffer` is reused.

We covered this [in the previous section](#) in the context of deserialization, the logic is the same: do not reuse a `JsonBuffer`.

6.4 Understand compiler errors

Long compiler errors

Compiler errors can be very difficult to understand, especially when the code uses a lot of templates, as ArduinoJson does, because a single error can produce a very long output.

Here is an example of output produced when compiling a program using ArduinoJson:

```
In file included from D:\libraries\ArduinoJson\...\JsonVariantBase.hpp:10:0,
                 from D:\libraries\ArduinoJson\...\JsonVariant.hpp:16,
                 from D:\libraries\ArduinoJson\...\JsonBuffer.hpp:15,
                 from D:\libraries\ArduinoJson\...\JsonParser.hpp:10,
                 from D:\libraries\ArduinoJson\...\JsonBufferBase.hpp:10,
                 from D:\libraries\ArduinoJson\...\DynamicJsonBuffer.hpp:10,
                 from D:\libraries\ArduinoJson\src\ArduinoJson.hpp:10,
                 from D:\libraries\ArduinoJson\src\ArduinoJson.h:10,
                 from D:\MyProgram\MyProgram.ino:7:
D:\libraries\ArduinoJson\...: In instantiation of 'ArduinoJson::JsonVariant...
D:\MyProgram\MyProgram.ino:132:27:   required from here
D:\libraries\ArduinoJson\...: error: invalid conversion from 'ArduinoJson:...
    return impl()->template as<T>();
                                   ^
In file included from D:\libraries\ArduinoJson\src\ArduinoJson.hpp:12:0,
                 from D:\libraries\ArduinoJson\src\ArduinoJson.h:10,
                 from D:\MyProgram\MyProgram.ino:7:
D:\libraries\ArduinoJson\...: In instantiation of 'ArduinoJson::Internals:...
D:\libraries\ArduinoJson\...:   required from 'ArduinoJson::Internals::List...
D:\libraries\ArduinoJson\...:   required from 'typename ArduinoJson::Intern...
D:\libraries\ArduinoJson\...:   required from 'typename ArduinoJson::TypeTr...
D:\libraries\ArduinoJson\...:   required from 'typename ArduinoJson::Intern...
D:\libraries\ArduinoJson\...:   required from 'ArduinoJson::JsonVariantCast...
D:\MyProgram\MyProgram.ino:132:27:   required from here
D:\libraries\ArduinoJson\...: error: 'equals' is not a member of 'ArduinoJs...
    if (Internals::StringTraits<TStringRef>::equals(key, it->key)) break;
                                   ^
```

I shorten the lines to 80 columns so that they could fit on the page, but the actual output had 533 columns!

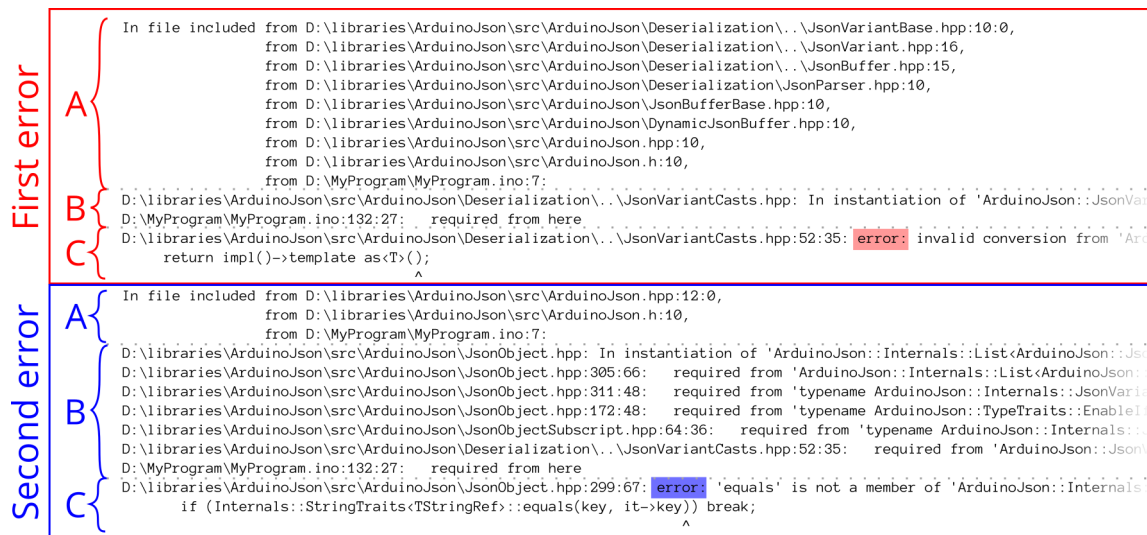
Most of the lines are referring to the ArduinoJson's source code, but this doesn't mean that the error is in the library.

How GCC presents errors

Most of the time, the compiler will produce more than one error or warning; it's essential to treat them one by one.

Start by finding the first line where the word `error:` or `warning:` appears, because it contains the location where the first problem was detected. If there are errors and warnings, choose the first of all because a warning often gives clues to understand an error.

Have a look at the picture below, where I highlighted the two errors from the listing above:



Highlighted parts of compiler output

As you can see, each error contains many lines within three groups:

- the “include stack,” which shows how the file was included,
- the “instantiation stack,” which shows how the type was instantiated,
- the actual error message.

You need to start your investigation from group C, as it contains the actual error message and shows where the error was detected. But this line shows the *effect*, not the *cause*, so we need to investigate further to find the cause.

The group B shows the steps that the compiler followed to build the erroneous line. It's a stack of instantiation, and it's very similar to the call stack that you see in a debugger. At the top of the stack is the most recent instantiation (the closest to the error), the second line is the instantiation that led to the first, etc. At the bottom of the stack is the original instantiation (the closest to the program), this is usually the root cause of the error.

Finally, group A shows how the erroneous file was included in the program. It shows which file included which, up to the root `.cpp` or `.ino`. It is rarely useful.

The first error in our example

When the output is complex, it can be useful to copy the content in a powerful text editor (I use Sublime Text). From there, we can remove the noise (long file path and namespace) and align the output to make it more readable.

Here is our first error (group C) after simplifying file path and namespaces:

```
JsonVariantCasts.hpp:52:35: error: invalid conversion from
  'JsonVariantAs<char*>::type {aka const char*}' to 'char*' [-fpermissive]
```

This line tells that the compiler is unhappy to convert a `const char*` into a `char*`. Indeed this conversion is forbidden, because it would lose the constness of the pointer. This error appears in `ArduinoJson`'s source code, but it's the *effect*, not the *cause*, we need to look at the instantiation stack (group B) to find the cause:

```
JsonVariantCasts.hpp: In instantiation of
  'JsonVariantCasts<TImpl>::operator T() const [with T = char*]':
MyProgram.ino:132:27:      required from here
```

It's usually easier to start at the bottom of the stack, because it points to a line in the calling program. In this case, the line 132 of `MyProgram.ino` contains:

```
char* value = obj[key];
```

Indeed, this line contains an error: `JsonObject` returns strings as `const char*`, not `char*`, hence the invalid conversion. We can fix this error by adding `const` in front of `char*`:

```
char* value = obj[key];
const char* value = obj[key];
```

This error was easy to find, no need to unroll the instantiation stack. Let's see the second.

The second error in our example

The second error message (group C) is:

```
JsonObject.hpp:299:error: 'equals' is not a member of 'StringTraits<const int&>'
  if (StringTraits<TStringRef>::equals(key, it->key)) break;
```

This line says that compiler is looking for a member named `equals` in the type `StringTraits<const int&>`, but this member doesn't exist. As the author of the library, I know what it means, but it's probably not your case. To understand what is happening, we need to look at the instantiation stack (group B):

```

JsonObject.hpp:      In instantiation of 'ArduinoJson::Internals::List<Arduino...
JsonObject.hpp:305:      required from 'ArduinoJson::Internals::List<JsonP...
JsonObject.hpp:311:      required from 'typename ArduinoJson::Internals::J...
JsonObject.hpp:172:      required from 'typename ArduinoJson::TypeTraits::...
JsonObjectSubscript.hpp:64: required from 'typename ArduinoJson::Internals::J...
JsonVariantCasts.hpp:52:  required from 'ArduinoJson::JsonVariantCasts<TImp...
MyProgram.ino:132:        required from here

```

The bottom of the stack is the same as for the first error, meaning that the same line generated two compiler errors. We already looked at this line, and it's not apparent why it has any relation to `StringTraits<const int&>`. To understand further, we need to unroll the stack: start from the bottom and climb up one line after the other.

The next instantiation (second from the bottom of group B) is:

```

JsonVariantCasts.hpp:52: required from
  'ArduinoJson::JsonVariantCasts<TImpl>::operator T() const
    [with T = char*;
      TImpl = ArduinoJson::JsonObjectSubscript<const int&>]'

```

Again, the meaning is probably very obscure to you, so let's move to the next instantiation (third from the bottom):

```

JsonObjectSubscript.hpp:64: required from
  'typename ArduinoJson::Internals::JsonVariantAs<T>::type
    ArduinoJson::JsonObjectSubscript<TKey>::as() const
    [with TValue = char*;
      TStringRef = const int&;
      ArduinoJson::Internals::JsonVariantAs<T>::type = const char*]'

```

This line starts to make sense. First, we see the `char*` and `const char*` that were causing the first error. Then, we see something very suspicious: `TStringRef = const int&`, which means that a template type named `TStringRef` has been set to `const int&`. From the name `TStringRef`, we can infer that a string reference is expected, but it is currently set to an `int` reference.

Let's look at the next instantiation (fourth from the bottom):

```
JsonObject.hpp:172: required from
'typename ArduinoJson::TypeTraits::EnableIf<
  (! ArduinoJson::TypeTraits::IsArray<TString>::value),
  typename ArduinoJson::Internals::JsonVariantAs<T>::type>::type
ArduinoJson::JsonObject::get(const TString&) const
  [with TValue = char*;
   TString = int;
   ArduinoJson::Internals::JsonVariantAs<T>::type = const char*]'
```

This line is very long, please focus on the highlighted part, where we see a call to `JsonObject::get()`. Let's go back to our original program; here is line 132 of `MyProgram.ino`:

```
char* value = obj[key];
```

During the compilation phase, this statement was transformed into the following:

```
char* value = obj.get<char*>(key);
```

The transformation occurred in the two first instantiations: `JsonVariantCasts.hpp:52` and `JsonObjectSubscript.hpp:64`.

But this call to `JsonObject::get(const TString&)` is problematic. Indeed, the compiler says `TString = int`, which means the template type `TString` was deduced as `int`, whereas a string is expected.

The bug is now right before our eyes: the variable `key` is not a string, it's an `int`!

Indeed a `JsonObject` is indexed by a string, and `ArduinoJson` doesn't support accessing a key-value pair by its index. To fix this error, we must change the type of the variable `key` to one of the supported string types: `const char*`, `String`...



Take away

Here are the things to remember when facing such error:

1. The error is not in the library, even if most lines refer to the library.
2. Copy the compiler output to a text editor and make it more readable.
3. Start by identifying the error messages (group C), they begin with `error:` or `warning:`.
4. Unroll the instantiation stack (group B) starting from the bottom.

6.5 Common error messages

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

Ambiguous overload for `operator=`

Most of the time you can rely on implicit casts, but there is one notable exception: when you convert a `JsonVariant` to a `String`. For example:

```
String ssid = network["ssid"];
ssid = network["ssid"];
```

The first line will compile but the second will fail with the following error:

```
error: ambiguous overload for 'operator='
      (operand types are 'String'
        and 'ArduinoJson::JsonObjectSubscript<const char*>')
```

The solution is to remove the ambiguity by explicitly casting the `JsonVariant` to a `String`:

```
ssid = network["ssid"];
ssid = network["ssid"].as<String>();
```

Conversion from `const char*` to `char*`

`ArduinoJson` returns keys and values as `const char*`. If you try to put these values into a `char*`, the compiler will issue an error (or a warning) like the following:

```
error: invalid conversion
      from 'ArduinoJson::Internals::JsonVariantAs<char*>::type {aka const char*}'
      to 'char*'
      [-fpermissive]
```

This error occurs with any of the following expression:

```
char* sensor = root["sensor"];
char* sensor = root["sensor"].as<char*>();

// in a function whose return type is char*
return root["sensor"].as<char*>();
```

To fix this error, replace `char*` by `const char*`;

```
const char* sensor = root["sensor"];
const char* sensor = root["sensor"].as<char*>();

// change the return type of the function to const char*
return root["sensor"].as<char*>();
```

Conversion from `const char*` to `int`

Let's say you have the following JSON to parse:

```
{
  "modules": [
    {
      "name": "hello",
      "id": 10
    },
    {
      "name": "world",
      "id": 20
    }
  ]
}
```

If you write the following program:

```
JsonObject& root = jb.parseObject(input);
JsonArray& modules = root["modules"];

int id = modules["hello"]["id"];
```

You'll get the following compilation error:


```
error: invalid conversion
  from 'const char*'
  to 'size_t {aka unsigned int}'
  [-fpermissive]
```

modules is an array of object; like any array it expects an integer argument to the subscript operator ([]), not a string.

To fix this error, you must use an integer:

```
int id = modules["hello"]["id"];
int id = modules[0]["id"];
```

equals is not a member of StringTraits<const int&>

This error occurs when you index a JsonObject with an integer instead of a string.

For example, it happens with the following code:

```
JsonObject& obj = jb.parseObject(input);
```

```
int key = 0;
const char* value = obj[key];
```

The compiler generates an error similar to the following:

```
error: 'equals' is not a member of
  'ArduinoJson::Internals::StringTraits<const int&, void>'
```

Indeed, a JsonObject can only be indexed by a string, like this:

```
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);
    Serial.println(kv.value.as<char*>());
}
```

Undefined reference to __cxa_guard_acquire and __cxa_guard_release

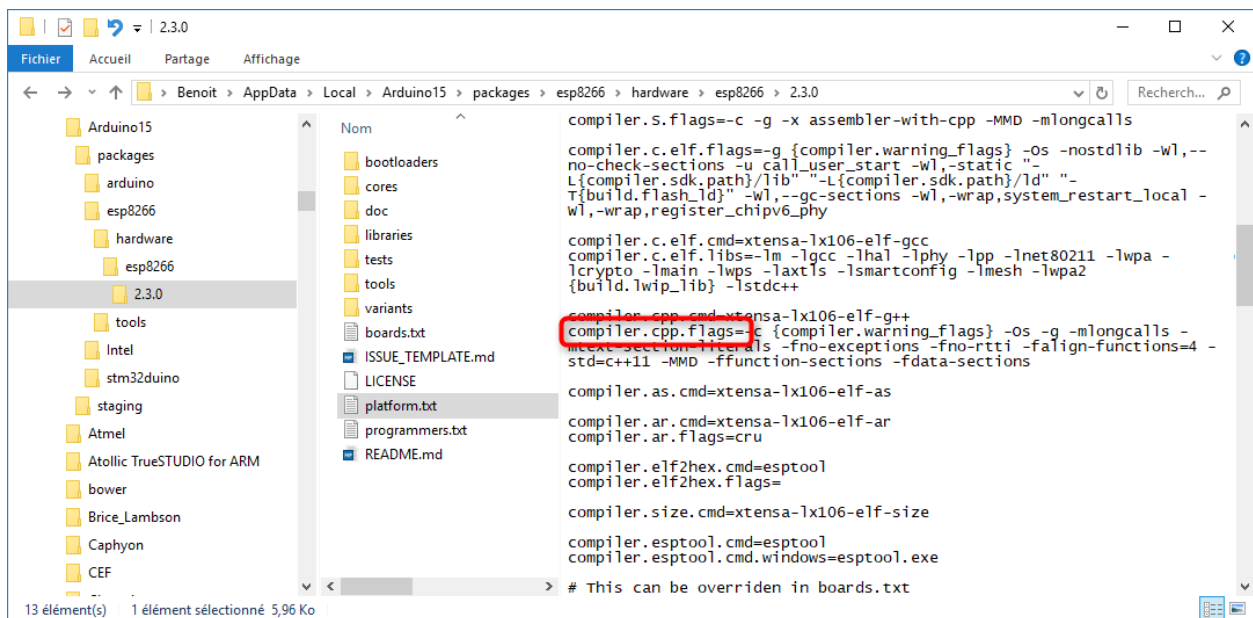
When the compiler is not properly configured, you can get the following errors:

```
In function `ArduinoJson::JsonArray::invalid()':
ArduinoJson/JsonArray.hpp:148: undefined reference to `__cxa_guard_acquire'
ArduinoJson/JsonArray.hpp:148: undefined reference to `__cxa_guard_release'
In function `ArduinoJson::JsonObject::invalid()':
ArduinoJson/JsonObject.hpp:131: undefined reference to `__cxa_guard_acquire'
ArduinoJson/JsonObject.hpp:131: undefined reference to `__cxa_guard_release'
collect2.exe: error: ld returned 1 exit status
```

To fix this, you need to pass a special flag to the compiler: `-fno-threadsafe-statics`.

If you use the Arduino IDE, you must add this flag to the `compiler.cpp.flags` in the `platform.txt` of the board you're using. This file is hard to find. On Windows, its located in the following folder:

`%LOCALAPPDATA%\Arduino15\packages\<brand>\hardware\<board>\<version>\platform.txt`



Compiler flags in platform.txt

6.6 Log

The problem

Consider a program that serializes a JSON document and sends it directly to its destination:

```
// Send a JSON document over the air  
obj.printTo(wifiClient);
```

On the one hand, we like this kind of code because it minimizes the 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 a another program that deserializes a JSON document directly from its origin:

```
// Receive a JSON document and parse it on-the-fly  
JsonObject& obj = jb.parseObject(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.

Don't worry. In both cases, there is a very simple solution.

Print decorator

The statement `obj.printTo(wifiClient)` is calling the function `JsonObject::printTo(Print&)`. This function takes an instance of `Print`, an abstract class that represent the concept of "output stream" in Arduino. We are free to create our own implementation of `Print` and pass it to `printTo()`.

For example, we can create a `Print` class whose job is to log and to delegate the work to another implementation. The `Print` class has three virtual functions, we only need to override them:

```
// An implementation of Print that logs and delegates to another implementation  
class PrintLogger : public Print {  
public:  
    // Constructs a PrintLogger attached to the specified target  
    PrintLogger(Print &target) : _target(target) {  
    }  
  
    // Writes a single byte  
    virtual size_t write(uint8_t c) {
```

```

    // Log to the serial
    Serial.write(c);
    // Delegate to the target
    return _target.write(c);
}

// Writes multiple bytes
virtual size_t write(const uint8_t *buffer, size_t size) {
    // Log to the serial
    Serial.write(buffer, size);
    // Delegate to the target
    return _target.write(buffer, size);
}

// Flushes temporary buffers
virtual size_t flush() {
    // Delegate to the target
    return _target.flush();
}

private:
    Print &_target;
};

```

This class can log the content sent to any implementation of `Print`, like `WifiClient`:

```

// Create a logger on top of the WifiClient
PrintLogger logger(wifiClient);

// Send a JSON document to the WifiClient and log at the same time
obj.printTo(logger);

```



The decorator pattern

`PrintLogger` is an implementation of a [design pattern](#) called “decorator”. This pattern allows to add behavior to an object without modifying its class. In our case, it gives the ability to log to any instance of `Print`.

Stream decorator

We can apply the decorator pattern to `parseObject()` too. `parseObject()` takes a reference to an instance of `Stream`, an abstract class representing the concept of “input stream” in Arduino.

The authors of Arduino decided to make `Stream` derive from `Print`. I think it is a bad idea because it forces every *input* stream to be an *output* stream too. Unfortunately, it's too late to change the `Stream` class, so we have to deal with it.

The `Stream` class adds three virtual functions to the ones inherited from `Print`, that's all we need to override:

```
class StreamLogger : public Stream {
public:
    // Constructs a StreamLogger attached to the specified target
    StreamLogger(Stream &target) : _target(target) {
    }

    // Pops the next byte from the stream
    virtual int read() {
        // Delegate to the target
        int c = _target.read();
        // Log if something was returned
        if (c > 0)
            Serial.print((char)c);
        // Pretend nothing happened
        return c;
    }

    // The following functions must be overridden to make the code compile,
    // but we don't need to log anything.
    virtual int available() { return _target.available(); }
    virtual void flush() { return _target.flush(); }
    virtual int peek() { return _target.peek(); }
    virtual size_t write(uint8_t c) { return _target.write(c); }
    virtual size_t write(const uint8_t *buffer, size_t size) {
        return _target.write(buffer, size);
    }

private:
    Stream &_target;
};
```

This class can log the content received from any implementation of `Stream`, like `WifiClient`:

```
// Create a logger on top of the WifiClient  
StreamLogger logger(wifiClient);  
  
// Receive a JSON document, log it and parse it on-the-fly  
JsonObject& obj = jb.parseObject(logger);
```

Now you can parse and view the JSON document at the same time.



Source files

You can find the code of these two classes in the folder `Decorators` of the zip file. Contrary to other code samples of the book, they are released under the term of the MIT License, so you are free to use them in your projects.

6.7 Ask for help

If none of the advice provided in this chapter helps, you should visit [ArduinoJson's FAQ](#), it covers more questions than this book and is frequently updated.

If you cannot find the answer in the FAQ, I recommend that you open a new [issue on GitHub](#). You may search existing issues first, but the FAQ already covers most.

When you write your issue, please take the time to write a good description. It's very important that you give the right amount of information. On the one hand, if you give too little (for example, just an error message without any context), I'll have to ask for more. On the other hand, if you provide too much information, I'll not be able to extract the signal from the noise.

The perfect description is composed of the following:

1. An 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 demonstrate the issue. It should have less than 50 lines. It's a good idea to test the MCVE on [wandbox.org](#) and share the link in the description of the issue. The process of writing an MCVE seems cumbersome, but it guarantees that the recipient (me, most likely) understands the problem quickly. Moreover, we often find the solution to our problem when we write the MCVE.

Respecting this rule shows that you care about the person receiving the request. Nobody wants to read gigantic code sample with dozens of suspicious dependencies. If you respect the time of others, they'll respect yours and give you a quick answer.

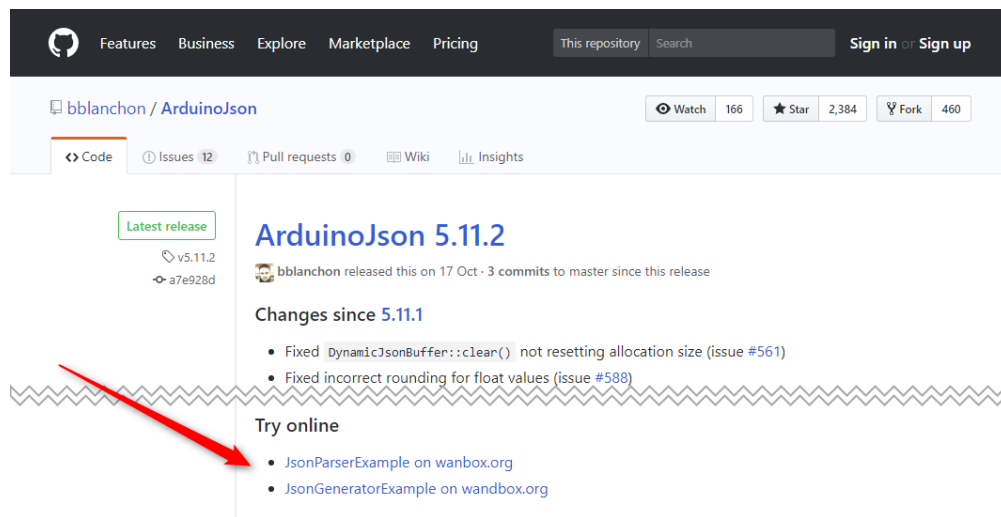
GitHub issues for ArduinoJson usually get an answer in less than 24 hour. A very high priority is given to actual bugs, but a very low priority is given to frequently asked questions.



MCVE on [wandbox.org](#)

If you want to write an MCVE on [wandbox.org](#), it's easier to start from an existing example, you can find links here:

- [ArduinoJson's home page](#)
- [GitHub releases page](#)



The screenshot shows the GitHub repository for `bblanchon / ArduinoJson`. The page includes navigation tabs for Code, Issues (12), Pull requests (0), Wiki, and Insights. The main content area displays the latest release, **ArduinoJson 5.11.2**, with a commit hash of `a7e928d`. Below the release information, a section titled "Changes since 5.11.1" lists two fixes: "Fixed `DynamicJsonBuffer::clear()` not resetting allocation size (issue #561)" and "Fixed incorrect rounding for float values (issue #588)". A red arrow points to the "Try online" section, which contains two links: "JsonParserExample on wanbox.org" and "JsonGeneratorExample on wandbox.org".

Links to wandbox.org

7. Case Studies

“

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

– Kent Beck

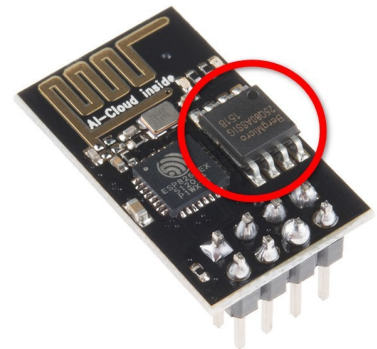
7.1 Configuration in SPIFFS

Presentation

For our first case study, we'll consider a program that has a global configuration object. When the program starts, it loads the configuration. The program saves the configuration whenever it changes.

The configuration is saved as a JSON document in a file. In our example, we'll consider the file-system SPIFFS, but you can easily adapt to any other file-system. SPIFFS allows storing files in a Flash memory connected to an SPI bus. Such memory chip is attached to every ESP8266.

The code for this case study is in the `SpiffsConfig` folder of the zip file.



The JSON document

Here is the layout of the configuration file:

```
{
  "access_points": [
    {
      "ssid": "SSID1",
      "passphrase": "PASSPHRASE1"
    },
    {
      "ssid": "SSID2",
      "passphrase": "PASSPHRASE2"
    }
  ],
  "server": {
    "host": "www.example.com",
    "path": "/resource",
    "username": "admin",
    "password": "secret"
  }
}
```

This document is composed of two parts:

1. a list of configurations for access points,
2. a configuration for a web service.

We assume the following constraints:

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

The configuration class

To persist the configuration in memory during the execution of the program, we need to create a structure that contains all the information.

We'll mirror the hierarchy of the JSON document in the configuration classes. While it is not mandatory to have the same hierarchy, it greatly simplifies the serialization code.

```
struct ApConfig {
    char ssid[32];
    char passphrase[64];
};

struct ServerConfig {
    char host[32];
    char path[32];
    char username[32];
    char password[32];
};

struct Config {
    static const int maxAccessPoints = 4;
    ApConfig accessPoint[maxAccessPoints];
    int accessPoints = 0;

    ServerConfig server;
};
```

In short, the Config class is the complete configuration of the program. It contains an array of four ApConfig to store the configuration of the access points, and a ServerConfig to store the configuration of the web service.

load() and save() members

To make sure that the JSON serialization is always in-sync with the content of each structure, we'll perform the mappings in member functions. We need two functions, one to load from a JSON document and another to save to a JSON document:

```
struct ApConfig {  
    char ssid[32];  
    char passphrase[64];  
  
    void load(const JsonObject &);  
    void save(JsonObject &) const;  
};  
  
// other structures also get their own load() and save()
```

To simplify the code of load() and save(), we only pass the part of the JSON document that is necessary. In the case of the ApConfig above, we only need the object for one access point:

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

There is a direct mapping from an ApConfig to a JsonObject because we decided to mirror the layout of the JSON document in the structures. The advantage is that this pattern keeps the related parts together; the drawback is that it couples the program with the JSON document: when one changes, you need to change the other.

Save an ApConfig into a JsonObject

Let's zoom into the save() function:

```
void ApConfig::save(JsonObject& obj) const {  
    obj["ssid"] = ssid;  
    obj["passphrase"] = passphrase;  
}
```

The role of this function is to save the content of the ApConfig into a JsonObject. It is marked a const because it doesn't modify the ApConfig. The code is very straightforward: it simply maps the members of the struct to the values in the JsonObject.

Load an ApConfig from a JsonObject

Now, let's see the other side:

```
void ApConfig::load(const JsonObject& obj) {
    strcpy(ssid, obj["ssid"] | "", sizeof(ssid));
    strcpy(passphrase, obj["passphrase"] | "", sizeof(passphrase));
}
```

It is the same thing in reverse: this function copies two strings from the `JsonObject` to the struct. It's a bit more complicated because we can't just copy the pointers returned by the `JsonObject`, we need to duplicate the content.

Safely copy strings from `JsonObject`

You're intuition was probably to write something like:

```
ssid = obj["ssid"];
```

But the member `ssid` is a `char[]`, so it cannot be assigned from a `const char*`, instead we need to copy the string returned by `ArduinoJson`. The C Standard library offers a function to copy a string to another: `strcpy()`.

```
strcpy(ssid, obj["ssid"]);
```

This code compiles and works most of the time, but it presents a security risk because `strcpy()` doesn't check the length of the destination. If the source is longer than expected, `strcpy()` blindly copies to the destination, causing a buffer overrun. The solution is to use `strncpy()` instead, and to specify the size of the destination:

```
strncpy(ssid, obj["ssid"], sizeof(ssid));
```

Again, this code compiles and works as long as the JSON contains a `ssid` key. Indeed, if `ssid` is missing `obj["ssid"]` returns a null, which is not supported by `strncpy()`. The solution is to provide a default value that is not null, using the `|` syntax of `ArduinoJson`:

```
strncpy(ssid, obj["ssid"] | "", sizeof(ssid));
```

I used an empty string as the default value, but we could use something else:

```
strncpy(ssid, obj["ssid"] | "default ssid", sizeof(ssid));
```

Save a Config to a `JsonObject`

We saw how to map an `ApConfig` to a `JsonObject`, and the process is the same for the `ServerConfig`. It's time to see how to build the complete JSON document described at the beginning. It is the role of the `save()` function of `Config`, the top-level configuration structure:

```

void Config::save(JsonObject& obj) const {
    // Add "server" object
    server.save(obj.createNestedObject("server"));

    // Add "access_points" array
    JSONArray& aps = obj.createNestedArray("access_points");

    // Add each access point in the array
    for(int i=0; i<accessPoints; i++)
        accessPoint[i].save(aps.createNestedObject());
}

```

As you can see, the Config structure delegates to its children the responsibility of saving themselves:

1. it calls `ServerConfig::save()` once to fill the server object,
2. it calls `ApConfig::save()` multiple times to save the configuration of each access point.

Load a Config from a JsonObject

Let's see the `load()` side:

```

void Config::load(const JsonObject& obj) {
    // Read "server" object
    server.load(obj["server"]);

    // Extract each access points
    JSONArray &aps = obj["access_points"];
    accessPoints = 0;
    for (JsonObject &ap : aps) {
        // Load the AP
        accessPoint[accessPoints].load(ap);

        // Increment AP count
        accessPoints++;

        // Is array full?
        if (accessPoints >= maxAccessPoints) break;
    }
}

```

Like the `save()` function, the `load()` function delegates to the children the responsibility of loading themselves. However, the function has a little bit of extra work because it needs to count the number of access points and make sure it doesn't exceed the capacity of the array.

Save configuration to a file

Up to now, we only wrote the code to map the configuration structures into `JsonObject`s and `JsonArrays`, but we didn't create any JSON document. Here is the function that produces the JSON document from the `Config` instance:

```
bool serializeConfig(const Config &config, Print& dst) {
    DynamicJsonBuffer jb(512);

    // Create an object
    JsonObject &root = jb.createObject();

    // Fill the object
    config.save(root);

    // Serialize JSON to file
    return root.prettyPrintTo(dst);
}
```

The function above is responsible for the serialization, but it doesn't create the file on disk. The following function creates the file and calls `serializeConfig()`:

```
bool saveFile(const char *filename, const Config &config) {
    // Open file for writing
    File file = SPIFFS.open(filename, "w");
    if (!file) {
        Serial.println(F("Failed to create configuration file"));
        return false;
    }

    // Serialize JSON to file
    bool success = serializeConfig(config, file);
    if (!success) {
        Serial.println(F("Failed to serialize configuration"));
        return false;
    }

    return true;
}
```

I decided to split the serialization and the file handling into different functions, because it keeps `serializeConfig()` independent from the underlying file-system. So, if you need to use another file system, you only need to modify `saveFile()`.

Read configuration from a file

Loading from a file is done similarly. One function is responsible for opening the file, and another does the deserialization:

```
bool deserializeConfig(Stream& src, Config &config) {
    DynamicJsonBuffer jb(1024);

    // Parse the JSON object in the file
    JsonObject &root = jb.parseObject(src);
    if (!root.success()) return false;

    config.load(root);
    return true;
}

bool loadFile(const char *filename, Config &config) {
    // Open file for reading
    File file = SPIFFS.open(filename, "r");

    // This may fail if the file is missing
    if (!file) {
        Serial.println(F("Failed to open config file"));
        return false;
    }

    // Parse the JSON object in the file
    bool success = deserializeConfig(file, config);

    // This may fail if the JSON is invalid
    if (!success) {
        Serial.println(F("Failed to deserialize configuration"));
        return false;
    }

    return true;
}
```

Choosing the JsonBuffer

In both cases, I used a DynamicJsonBuffer, but I could have used a StaticJsonBuffer because the capacity (1024 bytes) is significantly lower than the limit (4096 bytes on the ESP8266).

The size of the `JsonBuffer` for the deserialization is a lot bigger than the one for the serialization because it needs to copy the strings contained in the input stream. Remember that the file is a stream, each byte is read one by one and it's the `JsonBuffer` that is responsible for assembling the result.

I used the ArduinoJson Assistant to compute the capacities and rounded the results to the closest powers of two (512 and 1024). I created a sample JSON document with the maximum allowed: four access points and all string filled to the max.

Input		JsonBuffer size										
<pre>{ "ssid": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX" "passphrase": "XXX XXXXXXXXXXXXXXXXXXXX" }, "ssid": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX" "passphrase": "XXX XXXXXXXXXXXXXXXXXXXX" } "server": { "host": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX" "path": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX" "username": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX" "password": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX" }</pre>		<p>Expression</p> $JSON_ARRAY_SIZE(4) + 5 * JSON_OBJECT_SIZE(2) + JSON_OBJECT_SIZE(4)$ <p>Additional bytes for input duplication</p> <p>625</p> <table border="1"><thead><tr><th>Platform</th><th>Size</th></tr></thead><tbody><tr><td>AVR 8-bit</td><td>825</td></tr><tr><td>ESP8266</td><td>953</td></tr><tr><td>Visual Studio x86</td><td>1253</td></tr><tr><td>Visual Studio x64</td><td>1337</td></tr></tbody></table>	Platform	Size	AVR 8-bit	825	ESP8266	953	Visual Studio x86	1253	Visual Studio x64	1337
Platform	Size											
AVR 8-bit	825											
ESP8266	953											
Visual Studio x86	1253											
Visual Studio x64	1337											

Conclusion

There are many ways to write this program; other developers would do it differently. One could argue that the `save()` and `load()` functions should not be members of the configuration structures. Another could say that the configuration classes should not expose `JsonObject` in their public interface.

All these remarks make sense, however the perfect code doesn't exist, so we need to choose something that is practical and that makes sense in our context. I decided to present this implementation as a case study because:

1. It's scalable. You can add more configuration members and continue to apply the same `load()/save()` pattern to each child. As these functions are very short, the overall complexity will remain constant.
2. It's easy to maintain. The pattern groups the things that must change together in the same class. For example, if you add a member in `ServerConfig`, it's easy to figure out that `ServerConfig::load()` and `ServerConfig::save()` must change.

7.2 OpenWeatherMap on mkr1000

Presentation

For our second case study, we'll use a Genuino mkr1000 to collect weather forecast information from OpenWeatherMap. The program connects to a Wifi network, sends an HTTP request to `openweathermap.org`, and extracts the weather forecast from the response.

Although it looks similar to the Yahoo weather forecast that we saw [at the beginning of the book](#), we are not going to use the same technique. In the case of Yahoo, we were able to reduce the size of the JSON response, so that it could fit in memory. But in the case of OpenWeatherMap, it's not possible. The JSON response of OpenWeatherMap is so large that it's more interesting to deserialize it in pieces: one forecast after the other.



The code for this case study is in the `OpenWeatherMap` folder of the zip file. To run this program, you need a key for the API of OpenWeatherMap. You can get a key for free by creating an account on `openweathermap.org`.

In this section, I assume that you already read the [Yahoo Weather](#) and [Adafruit IO](#) examples at the beginning of the book.

OpenWeatherMap's API

OpenWeatherMap offers several services; in this example, we'll use the 5-day forecast. As the name suggests, it returns the weather information for the next 5 days. Each day is divided into periods of 3 hours (8 per day), so the response contains 40 forecasts.

To download the 5-day forecast, we need to send the following HTTP request:

```
GET /data/2.5/forecast?q=London&units=metric&appid=APIKEY HTTP/1.0
Host: api.openweathermap.org
Connection: close
```

APIKEY must be replaced with the API key linked to your account. London is just an example and can be replaced by another city.

Remark that we use `HTTP/1.0` instead of `HTTP/1.1` to disable Chunked Transfer Encoding.

The response to this request has the following shape:

HTTP/1.1 200 OK

Date: Thu, 30 Nov 2017 10:19:59 GMT

Content-Type: application/json; charset=utf-8

Content-Length: 14754

```
{"cod": "200", "message": 0.005, "cnt": 40, "list": [{ "dt": 1512043200, "main": { "temp": ...
```

The JSON response

The body of the response is a minified JSON document of 14 KB. This document looks like that:

```
{
  "cod": "200",
  "message": 0.005,
  "cnt": 40,
  "list": [
    {
      "dt": 1512043200,
      "main": {
        "temp": 3.33,
        "temp_min": 1.49,
        "temp_max": 3.33,
        "pressure": 1015.46,
        "sea_level": 1023.3,
        "grnd_level": 1015.46,
        "humidity": 79,
        "temp_kf": 1.84
      },
      "weather": [
        {
          "id": 600,
          "main": "Snow",
          "description": "light snow",
          "icon": "13d"
        }
      ],
      "clouds": {
        "all": 36
      },
      "wind": {
        "speed": 6.44,
        "deg": 324.504
      }
    }
  ]
}
```

```

    },
    "snow": {
      "3h": 0.036
    },
    "sys": {
      "pod": "d"
    },
    "dt_txt": "2017-11-30 12:00:00"
  }
],
"city": {
  "id": 2643743,
  "name": "London",
  "coord": {
    "lat": 51.5085,
    "lon": -0.1258
  },
  "country": "GB"
}
}

```

In the sample above, I preserved the hierarchical structure of the document, but I removed all forecast except one. In reality, the `list` array contains 40 objects.

Reducing memory usage

If we wanted to parse the entire document with `ArduinoJson` in one shot, it would require a `JsonBuffer` of 32 KB. It would not be possible in practice because the amount of RAM available in the Genuino mkr1000 is 32 KB.

To reduce the memory consumption, we'll parse only the parts of the document that are relevant to our project. We are only interested in the content of the `list` array which contains the 40 forecasts. So, instead of passing the complete document to `ArduinoJson`, we'll skip the beginning and parse only the array.

As the array is too big to be parsed in one shot, we'll parse each element one by one. It is easy because `ArduinoJson`'s parser stops after reading the closing brace (`}`), so the next character is either a comma (`,`) or a closing bracket (`]`). A comma means that there is another element to parse, whereas a closing bracket means that we reached the end of the array.

Jumping in the stream

The `Stream` class offers two convenient methods to jump to a known location:

- `Stream::find(char* target)` skips all characters until target is found,
- `Stream::findUntil(char* target, char* terminator)` skips all characters until target or terminator is found.

Unfortunately, both functions take `char*` instead of `const char*`, which prevent from passing a string literal directly, so we need to use a temporary variable. A [Pull Request](#) should fix this problem soon.

Assuming that `client` is our instance of `WifiClient`, here is how we can jump inside the `list` array:

```
char beginingOfList[] = "\"list\":[\"";
client.find(beginingOfList);
```

After `client.find()` returns, the next character is the opening brace (`{`) of the first forecast object. It is the object we want to parse, so it's time to use `ArduinoJson`:

```
StaticJsonBuffer<1024> jb;
JsonObject& forecast = jb.parseObject(client);
```

After `jb.parseObject()`, the next character is a comma (`,`), which separates the elements of the array. We need to skip this comma before parsing the next forecast object. We can repeat the process (parse object, skip comma, parse object, skip comma...), but we need to stop when we see the closing bracket (`]`). This is where we use `Stream::findUntil()`:

```
char comma[] = ",";
char endOfArray[] = "];";
bool found = client.findUntil(comma, endOfArray);
```

The return value of `Stream::findUntil()` indicates if target was found or if terminator was found, so we can use it as a stop condition for our loop.

Conclusion

This second case study was the opportunity to show a new technique to parse a big JSON document in pieces. It is a bit of extra work, but it's very simple. As an alternative, we could have used another library, like `json-streaming-parser`, but I'm convinced that the code would have been more complicated.

If you look at the source file, you'll see that the program uses `WifiClient` directly (without any HTTP abstraction) and uses only fixed memory allocation. We'll do it differently in the next case study.

7.3 Weather Underground on ESP8266

Presentation

For our third case study, we'll use an ESP8266 to collect weather forecast information from Weather Underground. Yes, this is the third example that downloads weather forecasts, but this kind of project is so common that I wanted to cover several cases.

As the program is similar to the previous one (connects to a Wifi network, sends an HTTP request and parse the response by pieces), we'll do things differently. Instead of using only stack memory, we'll use heap memory, and instead of making the HTTP request by hand, we'll use an abstraction.

The code for this case study is in the `WeatherUnderground` folder of the zip file. To run this program, you need a key for the API of WeatherUnderground. You can get a key for free by creating an account on wunderground.com.



Weather Underground's API

Weather Underground offers multiple services; in this example, we'll use the one called "forecast10day" which returns the weather for the next ten days.

There are two parts to the response:

1. `txt_forecast`, a textual summary of the weather conditions, contains two reports per day (20 elements in total);
2. `simpleforecast`, which includes weather parameters, contains one report per day.

In our case, we are only interested in a few parameters of the `simpleforecast`; therefore we'll ignore the `txt_forecast`.

To download the "forecast10day", we need to perform an HTTP GET with the following URL:

```
http://api.wunderground.com/api/APIKEY/forecast10day/q/CA/San_Francisco.json
```

`APIKEY` must be replaced by your own API key. `CA/San_Francisco` can be replaced by another city.

HTTP client

This time, we won't perform the HTTP request manually. Instead, we'll use the library `ESP8266HTTPClient`, which is part of the Arduino Core for ESP8266.

Here is the code to send the request:

```
HTTPClient http;  
http.begin(url);  
int status = http.GET();
```

Then, we need an instance of Stream that we can pass to ArduinoJson:

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

We can now use response to read the HTTP response.

The JSON response

The response contains an 18 KB JSON document that is **not minified**. This document looks like that:

```
{  
  "response": {  
    "version": "0.1",  
    "termsofService": "http://www.wunderground.com/weather/api/d/terms.html",  
    "features": {  
      "forecast10day": 1  
    }  
  },  
  "forecast": {  
    "txt_forecast": {  
      "date": "3:59 AM PST",  
      "forecastday": [  
        {  
          "period": 0,  
          "icon": "clear",  
          "icon_url": "http://icons.wxug.com/i/c/k/clear.gif",  
          "title": "Thursday",  
          "fcttext": "Sunny. High 61F. Winds NNW at 5 to 10 mph.",  
          "fcttext_metric": "Sunny. High 16C. Winds NNW at 10 to 15 km/h.",  
          "pop": "0"  
        }  
      ]  
    },  
    "simpleforecast": {  
      "forecastday": [  
        {
```

```
"date": {
  "epoch": "1512097200",
  "pretty": "7:00 PM PST on November 30, 2017",
  "day": 30,
  "month": 11,
  "year": 2017,
  "hour": 19,
  "min": "00",
  "sec": 0,
  "monthname": "November",
  "monthname_short": "Nov",
  "weekday_short": "Thu",
  "weekday": "Thursday"
},
"period": 1,
"high": {
  "fahrenheit": "61",
  "celsius": "16"
},
"low": {
  "fahrenheit": "48",
  "celsius": "9"
},
"conditions": "Clear",
"icon": "clear",
"icon_url": "http://icons.wxug.com/i/c/k/clear.gif",
"skyicon": "",
"pop": 0,
"maxwind": {
  "mph": 10,
  "kph": 16,
  "dir": "NNW",
  "degrees": 344
},
"avewind": {
  "mph": 9,
  "kph": 14,
  "dir": "NNW",
  "degrees": 344
},
"avehumidity": 67,
"maxhumidity": 0,
```



```

        "minhumidity": 0
    }
]
}
}
}

```

In the sample above, I preserved the hierarchical structure of the document, but I kept only one forecast. In reality, `forecast.txt_forecast.forecastday` contains 20 objects and `forecast.simpleforecast.forecast` contains 10 objects. I also removed several irrelevant fields from the `simpleforecast` to make it shorter.

Reducing memory usage

If we wanted to parse the entire response with `ArduinoJson` in one shot, it would require a `JsonBuffer` of 25 KB. It could work since the ESP8266 has 96 KB of RAM, but we won't do that because I want to show you how to parse the response in pieces.

Instead, we'll use the same technique as with `OpenWeatherMap`:

1. ignore the beginning of the document and jump directly into the first object in the array
`forecast.simpleforecast.forecastday`,
2. call `parseObject()`,
3. skip the comma,
4. repeat until the closing bracket (`]`) is found.

Jumping in the stream

In the previous section, I presented `find()` and `findUntil()` and said they had an issue because they take `char*` instead of `const char*`. Fortunately, the authors of the Arduino Core for ESP8266 fixed this issue, so we don't have to use a temporary variable.

Here are the two functions:

- `Stream::find(const char* target)` skips all characters until `target` is found.
- `Stream::findUntil(const char* target, const char* terminator)` skips all characters until `target` or `terminator` is found.

Here is how we can jump to the `simpleforecast` object:

```
response.find("\simpleforecast");
```

We are now at the beginning of simpleforecast object, so we can jump to the forecastday array:

```
response.find("\forecastday");
```

We are now at the beginning of the forecastday array, so we can jump to the first object in the array:

```
response.find("[");
```

We need to do the jump with three calls because there can be a variable number of spaces between the characters. It would have been much simpler with a minified document because one call to find() would have been sufficient:

```
response.find("\simpleforecast\":{\"forecastday\":[");
```

Now that we are at the right location in the stream, we can parse the JSON object:

```
DynamicJsonBuffer jb(2048);  
JsonObject &obj = jb.parseObject(response);
```

Finally, to skip the comma and detect the closing bracket, we use findUntil(), as before:

```
bool found = response.findUntil(",", "]);
```

Conclusion

This third case study was very similar to the second but showed how to apply the same technique when the JSON document is not minified. It also demonstrated how to use the library ESP8266HTTPClient with ArduinoJson.

I promise, no more weather project in this book ;-)

7.4 JSON-RPC with Kodi

Presentation

For our fourth case study, we'll create a remote for Kodi. Kodi (formerly known as XBMC) is a software media player. It allows playing videos and music on a computer. It is the most popular software to create an HTPC (Home Theater Personal Computer) because it has a full-screen GUI and can be controlled with a remote.

Kodi can also be controlled remotely via a network connection; we'll use this feature to control it from our program. Kodi 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 (for example, in Jeedom, the home-automation project), so you can reuse most of the code that we see in this project.

To spice the thing a little, I'll use an Arduino UNO with an Ethernet Shield. This restriction makes the task more difficult because the UNO has only 2KB of RAM, so we need to be careful not to waste it.

The code for this case study is in the `KodiRemote` folder of the zip file. To run this program, you need to run an instance of Kodi and check the box "Allow remote control via HTTP" in the settings.



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 of the screen.

The object contains two other values, `jsonrpc` and `id`, which are imposed by the JSON-RPC protocol.

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 `GUI.ShowNotification` procedure. In this case, `result` is a simple string, but most of the time, it is a JSON object.

If a call fails, the server removes `result` and adds an `error` object:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32601,
    "message": "Method not found."
  },
  "id": 1
}
```

A JSON-RPC framework

To implement JSON-RPC, we create three classes:

- `JsonRpcRequest` represents a JSON-RPC request,
- `JsonRpcResponse` represents a JSON-RPC response,
- `JsonRpcClient` sends `JsonRpcRequests` and receives `JsonRpcResponses`.

As the RAM is scarce in this project, we make sure that only the `JsonRpcRequest` or the `JsonRpcResponse` is in memory, but not both at the same time. To implement this, we'll use the following steps:

1. create the request,
2. send the request,
3. destroy the request,
4. receive the response,
5. destroy the response.

This technique requires extra work and discipline but reduces the memory consumption in half.

JsonRpcRequest

We start by creating the class `JsonRpcRequest` which represents a JSON-RPC request. The class owns a `JsonBuffer` and a reference to the `JsonObject`. It also exposes the `param` object to the caller:

```
class JsonRpcRequest {  
    StaticJsonBuffer<512> _jb;  
    JsonObject &_root;  
  
public:  
    JsonObject &params;  
}
```

I used a `StaticJsonBuffer` with a fixed size of 512 bytes because I know it would be unreasonable to ask more from a poor little UNO, but you can use a `DynamicJsonBuffer` if you're using a bigger processor.

Embedding the `JsonBuffer` inside the request ensure that their lifetimes are identical. We are sure that every byte is released when we destroy the request.

The constructor of `JsonRpcRequest` takes the name of the procedure to call. It is responsible for creating the skeleton of the request. It has to do some gymnastics to fill all the members of the class in the right order. Because `_root` and `params` are references, they need to be initialized in the member initializers.

Here is the constructor:

```

JsonRpcRequest::JsonRpcRequest(const char *method)
    : _jb(), // JsonBuffer must be created first
      _root(_jb.createObject()), // then we can create the root
      params(_root.createNestedObject("params")) { // then the nested object
    _root["method"] = method;
    _root["jsonrpc"] = "2.0";
    _root["id"] = 1;
}

```

Finally, the class exposes two functions to serialize the request, they will be used by the `JsonRpcClient`:

```

// Computes Content-Length
size_t JsonRpcRequest::length() const {
    return _root.measureLength();
}

// Serializes the request
size_t JsonRpcRequest::printTo(Print &client) const {
    return _root.printTo(client);
}

```

JsonRpcResponse

We use a very similar pattern for the class `JsonRpcResponse` which represents a JSON-RPC response. This class owns a `JsonBuffer` and exposes `result` and `error`:

```

class JsonRpcResponse {
    StaticJsonBuffer<512> _jb;

public:
    JsonVariant result;
    JsonVariant error;
};

```

This class has only one function which is called by the `JsonRpcClient` when the response arrives:

```
bool JsonRpcResponse::parse(Stream &stream) {
    JsonObject &root = _jb.parse(stream);
    result = root["result"];
    error = root["error"];
    return root.success();
}
```

JsonRpcClient

We can now create the last piece of our JSON-RPC framework: the `JsonRpcClient`. This class is responsible for sending requests and receiving responses over HTTP. It owns an instance of `EthernetClient`, and saves the hostname and port to be able to reconnect at any time:

```
class JsonRpcClient {
public:
    JsonRpcClient(const char *host, short port) : _host(host), _port(port) {

        // Sends a JSON-RPC Request
        bool send(const JsonRpcRequest &req);

        // Receives a JSON-RPC Response
        bool recv(JsonRpcResponse &res);

private:
    EthernetClient _client;
    const char *_host;
    short _port;
};
```

`JsonRpcClient` exposes two functions that the calling program uses to send requests and receive responses. The two functions are separated because, as we said, we don't want to have the `JsonRpcRequest` and the `JsonRpcResponse` in memory at the same time. That would not be possible with a signature like:

```
bool call(const JsonRpcRequest &req, JsonRpcResponse &res);
```

The `send()` function is responsible for establishing the connection with the server, and for sending the HTTP request:

```
bool JsonRpcClient::send(const JsonRpcRequest &req) {  
    // Connect with server  
    _client.connect(_host, _port);  
  
    // Send the HTTP headers  
    _client.println(F("POST /jsonrpc HTTP/1.0"));  
    _client.println(F("Content-Type: application/json"));  
    _client.print(F("Content-Length: "));  
    _client.println(req.length());  
    _client.println();  
  
    // Send JSON document in body  
    req.printTo(_client);  
  
    return true;  
}
```

The `recv()` function is responsible for skipping the response's HTTP headers and for extracting the JSON body:

```
bool JsonRpcClient::recv(JsonRpcResponse &res) {  
    // Skip HTTP headers  
    char endOfHeaders[] = "\r\n\r\n";  
    _client.find(endOfHeaders);  
  
    // Parse body  
    return res.parse(_client);  
}
```

I removed the error checking from the two snippets above, please see the source files for the complete code.

Send a notification to Kodi

To display a popup on Kodi's screen, we need to send the following request:


```
{
  "jsonrpc": "2.0",
  "method": "GUI.ShowNotification",
  "params": {
    "title": "Title of the notification",
    "message": "Content of the notification"
  },
  "id": 1
}
```

And, in return, we expect the following response:

```
{
  "jsonrpc": "2.0",
  "result": "OK",
  "id": 1
}
```

Let's use our new JSON-RPC framework to do this:

```
// Create the JSON-RPC client
JsonRpcClient client(host, port);

// This is the scope of the Request object
{
  // Create the request, passing the procedure name
  JsonRpcRequest req("GUI.ShowNotification");

  // Add the two parameters
  req.params["title"] = title;
  req.params["message"] = message;

  // Send the request
  client.send(req);
}

// This is the scope of the Response object
{
  // Create empty response
  JsonRpcResponse res;

  // Read response
```

```
client.recv(res);

// Is this the expected result?
if (res.result != "OK") {
    Serial.println(F("ERROR!"));
    // Dump the error object to the Serial
    res.error.prettyPrintTo(Serial);
}
}
```

As you see, we use a scope to limit the lifetime of the `JsonRpcRequest`. It's not mandatory to wrap the scope of `JsonRpcResponse` between braces, but I did it because it adds symmetry to the code.

Get properties from Kodi

The procedure `Application.GetProperties` can retrieve information, we'll use it to get the running version of Kodi. This procedure takes one parameter: an array containing the names for the properties to read. In our case, there are two properties: `name` and `version`.

We need to send the following request:

```
{
  "jsonrpc": "2.0",
  "method": "Application.GetProperties",
  "params": {
    "properties": [
      "name",
      "version"
    ]
  },
  "id": 1
}
```

And here is the expected response:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "name": "Kodi",
    "version": {
      "major": 17,
      "minor": 6,
      "revision": "20171114-a9a7a20",
      "tag": "stable"
    }
  }
}
```

Let's use our framework again:

```
// Create the JSON-RPC client
JsonRpcClient client(host, port);

// This is the scope of the Request object
{
  // Create the request
  JsonRpcRequest req("Application.GetProperties");

  // Add the "properties" array, with the two names
  JsonArray &properties = req.params.createNestedArray("properties");
  properties.add("name");
  properties.add("version");

  // Send the request
  client.send(req);
}

// This is the scope of the Response object
{
  // Create an empty response
  JsonRpcResponse res;

  // Read response
  client.recv(res);

  // Print "Kodi 17.6 stable"
```

```
Serial.print(res.result["name"].as<char*>());  
Serial.print(" ");  
Serial.print(res.result["version"]["major"].as<int>());  
Serial.print(" ");  
Serial.print(res.result["version"]["minor"].as<int>());  
Serial.print(" ");  
Serial.println(res.result["version"]["tag"].as<char*>());  
}
```

Conclusion

The goal of this case study was to teach two things:

1. how to do JSON-RPC with ArduinoJson,
2. how to control the lifetime of objects.

Of course, the second goal made the first more complicated, but the effort was reasonable. Once you understand the pattern, the code is simple, and you can implement virtually any remote procedure call with the minimal amount of memory.

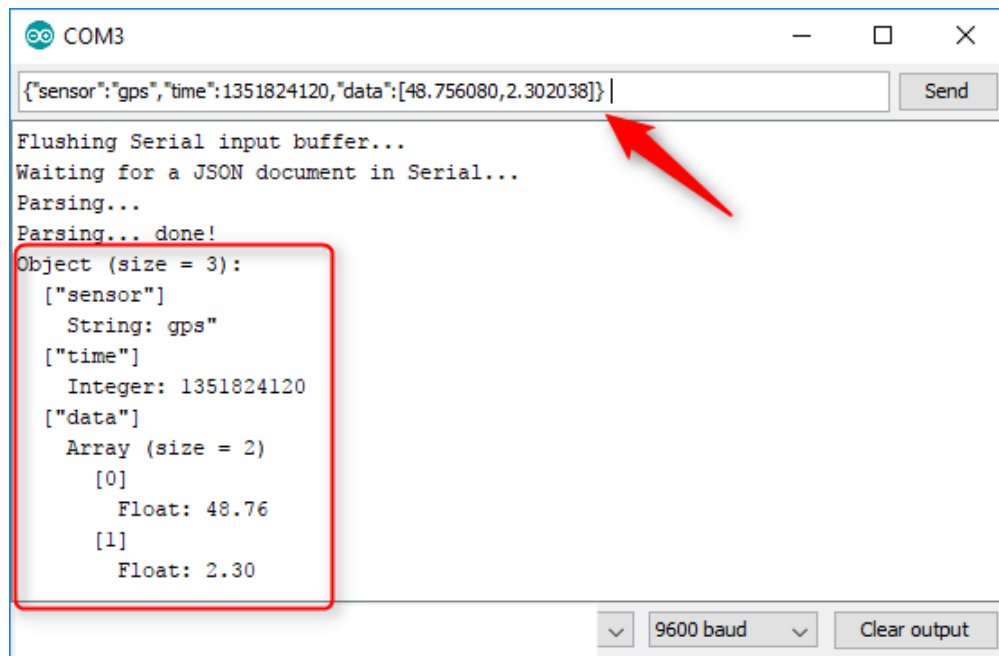
If you look at the source files, you'll see that I created a class `KodiClient` that provides the abstraction on top of `JsonRpcClient`, this class can easily be extended to add more procedures, for example, to control the playback.

Another difference in the source files is that I added an optimization: the TCP connections to the server are reused, using the “keep-alive” mode. This greatly improves the performance if you need to call several procedures in a row. To do that, the `KodiClient` reuses the same `JsonRpcClient` for each call.

7.5 Recursive analyzer

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. The goal is to demonstrate how to recursively scan a `JsonObject` or a `JsonArray`. This case study is also an opportunity to see how we can read a JSON document from the serial port



The source code of this program is in the *Analyzer* folder of the zip file.

Read from the serial port

`Serial` is an instance of the class `HardwareSerial`, which derives from `Stream`. As such, you can directly pass it to `ArduinoJson`:

```
DynamicJsonBuffer jb;  
JsonVariant root = jb.parse(Serial);
```

As you see, we call neither `parseObject()` nor `parseArray()` because we don't know what's in the JSON document, instead we call `parse()` which can handle both cases.

However, if we just do just that, the program will periodically report that the parsing fails. The failures are caused by:

1. a time-out waiting for the input document.
2. a trailing line break (`'\n'`) or carriage return (`\r`) sent by the Arduino Serial Monitor.

To solve the time-out issue, we just need to add a loop that waits for available characters:

```
// Loop until there is data waiting to be read  
while (!Serial.available())  
    delay(50);
```

To solve the trailing characters issue, we need to add a loop that flushes every character from the serial port:

```
// Read all remaining bytes from the serial port  
while (Serial.available())  
    Serial.read();
```

This last part is only required because we use the Arduino Serial Monitor. If you were using a serial port to communicate between two Arduinos, you wouldn't need to add this flushing loop.

Test 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 content of the values in 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 type of values:

1. boolean: `bool`
2. integral: `long` (but `int` and others would match too)
3. floating point: `double` (but `float` would match too)
4. string: `const char*`
5. object: `JsonObject&`
6. array: `JsonArray&`

We'll limit the responsibility of `dump(JsonVariant)` to the detection of the type, and we'll delegate the work of printing the value to overloads. The code is therefore just a sequence of `if` statement:

```

void dump(JsonVariant variant) {
    // if boolean, then call the overload for boolean
    if (variant.is<bool>())
        dump(variant.as<bool>());

    // if integral, then call the overload for integral
    else if (variant.is<long>())
        dump(variant.as<long>());

    // if floating point, then call the overload for floating point
    else if (variant.is<double>())
        dump(variant.as<double>());

    // if string, then call the overload for string
    else if (variant.is<char *>())
        dump(variant.as<char *>());

    // if object, then call the overload for object
    else if (variant.is<JsonObject>())
        dump(variant.as<JsonObject>());

    // if array, then call the overload for array
    else if (variant.is<JsonArray>())
        dump(variant.as<JsonArray>());

    // none of the above, then it's undefined
    else
        Serial.println(F("Undefined"));
}

```



Order matters

It's important to test long before double, because an integral can always be stored in a floating-point. In other words, `is<long>()` implies `is<double>()`.

Print values

Printing simple values is straightforward:

```
void dump(bool value) {
    Serial.print(F("Bool: "));
    Serial.print(value);
    Serial.println('');
}

void dump(long value) {
    Serial.print(F("Integer: "));
    Serial.println(value);
}

void dump(double value) {
    Serial.print(F("Float: "));
    Serial.println(value);
}

void dump(const char *str) {
    Serial.print(F("String: "));
    Serial.print(str);
    Serial.println('');
}
```

Printing objects requires a loop and a recursion:

```
void dump(const JsonObject &obj) {
    Serial.print(F("Object: "));

    // Iterate though all key-value pairs
    for (JsonPair kvp : obj) {
        // Print the key
        Serial.println(kvp.key);

        // Print the value
        dump(kvp.value); // <- RECURSION
    }
}
```

And so does printing an array:


```
void dump(const JSONArray &arr) {
    Serial.print(F("Array: "));

    int index = 0;
    // Iterate though all elements
    for (JsonVariant value : arr) {
        // Print the index
        Serial.println(index);

        // Print the value
        dump(value); // <- RECURSION

        index++;
    }
}
```



Prefer ranged-based for loop

We use the syntax `for(value:arr)` instead of `for(i=0;i<arr.size();++i)` because it's much faster. Indeed, it prevents from calling `arr[i]`, which needs to walk the linked-list (complexity $O(n)$).

Conclusion

It was by far the simplest of our case studies, but I'm sure it will be helpful to many readers as the recursive part can be tricky if you are not familiar with the technique.

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.

8. Conclusion

It's already the end of the book. I hope you enjoyed reading it as much as I enjoyed writing it. More importantly, I hope you learn many things about Arduino, C++, and programming in general. Please tell me what you thought of the book at book@arduinojson.org, I'll be happy to hear from you.

As I wrote this book, I saw ArduinoJson with the eye of a new user, which made me see many areas of improvement and simplification. You can be sure that ArduinoJson will evolve in the coming year and will become more versatile and easier to use. This book will be updated accordingly, and you'll be able to download the new version for free.

Again, thank you very much for buying this book. This act encourages the development of high-quality libraries. By providing a (modest) source of revenue for open-source developers like me, you ensure that the libraries that you rely on are continuously improved and won't be left abandoned after a year.

Sincerely, Benoit Blanchon



Satisfaction survey

Please take a minute to answer a short survey.

Go to arduinojson.poll daddy.com/s/book