Mastering ArduinoJson 7
Efficient JSON serialization for embedded C++
# Contents

## 1 Introduction

1.1 About this book .......................................................... 2  
1.1.1 Overview ................................................................. 2  
1.1.2 Code samples .......................................................... 2  
1.1.3 What changed since Mastering ArduinoJson 6 ................. 3  
1.2 Introduction to JSON .................................................. 4  
1.2.1 What is JSON? .......................................................... 4  
1.2.2 What is serialization? ................................................ 5  
1.2.3 What can you do with JSON? ....................................... 5  
1.2.4 History of JSON ...................................................... 8  
1.2.5 Why is JSON so popular? ............................................ 9  
1.2.6 The JSON syntax ..................................................... 9  
1.2.7 Binary data in JSON ................................................. 13  
1.2.8 Comments in JSON .................................................. 13  
1.3 Introduction to ArduinoJson ......................................... 15  
1.3.1 What ArduinoJson is ............................................... 15  
1.3.2 What ArduinoJson is not .......................................... 15  
1.3.3 What makes ArduinoJson different? ............................. 16  
1.3.4 Does size matter? ................................................... 18  
1.3.5 What are the alternatives to ArduinoJson? ...................... 18  
1.3.6 How to install ArduinoJson ...................................... 20  
1.3.7 The examples .......................................................... 25  
1.4 Summary ................................................................. 27  

## 2 The missing C++ course

2.1 Why a C++ course? ..................................................... 29  
2.2 Harvard and von Neumann architectures ............................ 31  
2.3 Stack, heap, and globals .............................................. 33  
2.3.1 Globals ................................................................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.2</td>
<td>Heap</td>
<td>35</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Stack</td>
<td>36</td>
</tr>
<tr>
<td>2.4</td>
<td>Pointers</td>
<td>38</td>
</tr>
<tr>
<td>2.4.1</td>
<td>What is a pointer?</td>
<td>38</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Dereferencing a pointer</td>
<td>38</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Pointers and arrays</td>
<td>39</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Taking the address of a variable</td>
<td>40</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Pointer to class and struct</td>
<td>40</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Pointer to constant</td>
<td>41</td>
</tr>
<tr>
<td>2.4.7</td>
<td>The null pointer</td>
<td>43</td>
</tr>
<tr>
<td>2.4.8</td>
<td>Why use pointers?</td>
<td>44</td>
</tr>
<tr>
<td>2.5</td>
<td>Memory management</td>
<td>45</td>
</tr>
<tr>
<td>2.5.1</td>
<td>malloc() and free()</td>
<td>45</td>
</tr>
<tr>
<td>2.5.2</td>
<td>new and delete</td>
<td>45</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Smart pointers</td>
<td>46</td>
</tr>
<tr>
<td>2.5.4</td>
<td>RAII</td>
<td>48</td>
</tr>
<tr>
<td>2.6</td>
<td>References</td>
<td>49</td>
</tr>
<tr>
<td>2.6.1</td>
<td>What is a reference?</td>
<td>49</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Differences with pointers</td>
<td>49</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Reference to constant</td>
<td>50</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Rules of references</td>
<td>51</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Common problems</td>
<td>51</td>
</tr>
<tr>
<td>2.6.6</td>
<td>Usage for references</td>
<td>52</td>
</tr>
<tr>
<td>2.7</td>
<td>Strings</td>
<td>53</td>
</tr>
<tr>
<td>2.7.1</td>
<td>How are the strings stored?</td>
<td>53</td>
</tr>
<tr>
<td>2.7.2</td>
<td>String literals in RAM</td>
<td>53</td>
</tr>
<tr>
<td>2.7.3</td>
<td>String literals in Flash</td>
<td>54</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Pointer to the “globals” section</td>
<td>55</td>
</tr>
<tr>
<td>2.7.5</td>
<td>Mutable string in “globals”</td>
<td>56</td>
</tr>
<tr>
<td>2.7.6</td>
<td>A copy in the stack</td>
<td>57</td>
</tr>
<tr>
<td>2.7.7</td>
<td>A copy in the heap</td>
<td>58</td>
</tr>
<tr>
<td>2.7.8</td>
<td>A word about the String class</td>
<td>59</td>
</tr>
<tr>
<td>2.7.9</td>
<td>Passing strings to functions</td>
<td>60</td>
</tr>
<tr>
<td>2.8</td>
<td>Summary</td>
<td>62</td>
</tr>
</tbody>
</table>

3 Deserialize with ArduinoJson

3.1 The example of this chapter | 65
3.2 Deserializing an object | 66
3.2.1 The JSON document | 66
3.2.2 Deserializing the JSON document | 66
3.3 Extracting values from an object ......................................................... 68
  3.3.1 Extracting values ................................................................. 68
  3.3.2 Explicit casts ................................................................. 68
  3.3.3 When values are missing ...................................................... 69
  3.3.4 Changing the default value .................................................... 70
3.4 Inspecting an unknown object ......................................................... 71
  3.4.1 Getting a reference to the object ......................................... 71
  3.4.2 Enumerating the keys ....................................................... 72
  3.4.3 Detecting the type of value ................................................ 72
  3.4.4 Variant types and C++ types ............................................. 73
  3.4.5 Testing if a key exists in an object ..................................... 73
3.5 Deserializing an array ................................................................. 75
  3.5.1 The JSON document ......................................................... 75
  3.5.2 Parsing the array ............................................................ 75
3.6 Extracting values from an array ...................................................... 77
  3.6.1 Retrieving elements by index ............................................. 77
  3.6.2 Alternative syntaxes ....................................................... 77
  3.6.3 When complex values are missing ....................................... 78
3.7 Inspecting an unknown array ........................................................... 80
  3.7.1 Getting a reference to the array ......................................... 80
  3.7.2 Number of elements in an array ......................................... 80
  3.7.3 Iteration ........................................................................... 81
  3.7.4 Detecting the type of an element ......................................... 81
3.8 Reading from a stream ..................................................................... 83
  3.8.1 Reading from a file ........................................................... 83
  3.8.2 Reading from an HTTP response ......................................... 84
3.9 The ArduinoJson Assistant ............................................................... 92
  3.9.1 Step 1: Configuration ......................................................... 93
  3.9.2 Step 2: JSON .................................................................... 94
  3.9.3 Step 3: Program ............................................................... 95
3.10 Summary ......................................................................................... 96

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

6.1 Introduction ................................................. 172
6.2 Variants ...................................................... 173
6.3 Integers ....................................................... 175
6.4 String .......................................................... 177
   6.4.1 String nodes ............................................. 177
   6.4.2 String adapters ......................................... 178
   6.4.3 Variable-length arrays ................................ 178
6.5 Arrays and objects ........................................... 180
6.6 Document tree ............................................... 182
6.7 Slot pool ...................................................... 184
6.8 The resource manager ....................................... 186
6.9 Smart pointers ............................................... 187
   6.9.1 JsonVariant ............................................. 187
   6.9.2 Unbound JsonVariant ................................... 187
   6.9.3 JsonVariantConst ........................................ 188
   6.9.4 JsonArray and JsonObject ............................... 189
6.10 Comparison operators ........................................ 190
6.11 Converters .................................................... 191
6.12 Proxies ....................................................... 193
6.13 Deserializers ............................................... 195
   6.13.1 Reading from various types ............................ 195
   6.13.2 Reading one character at a time ...................... 196
   6.13.3 Nesting limit .......................................... 196
   6.13.4 Escape sequences ..................................... 198
   6.13.5 Filtering .............................................. 199
   6.13.6 String-to-float conversion ............................ 199
6.14 Serializers ................................................... 200
   6.14.1 Writing to various types .............................. 200
   6.14.2 Float-to-string conversion ............................. 201
6.15 Namespaces .................................................. 202
6.16 Aggregated header ........................................... 203
6.17 Summary ......................................................... 204

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

8 Case Studies ...................................................... 238
  8.1 JSON Configuration File .................................... 239
    8.1.1 Presentation ......................................... 239
    8.1.2 The JSON document .................................. 239
    8.1.3 The configuration class .............................. 240
    8.1.4 Converters ............................................. 241
Chapter 3

Deserialize with ArduinoJson

“It is not the language that makes programs appear simple. It is the programmer that makes the language appear simple!”
3.1 The example of this chapter

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

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

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

There are several versions of GitHub’s API; we’ll use the latest one: the GraphQL API (or v4). We’ll use this one because it allows us to get all the information we need with only one query. It also returns much smaller responses than v3, which is appreciable for embedded software.

To run this example, you’ll need a user account on GitHub and a personal access token. Don’t worry; we’ll see that later.

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

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

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

3.2.1 The JSON document

Our example is the repository information for ArduinoJson:

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

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

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

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

3.2.2 Deserializing the JSON document

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

```cpp
JsonDocument doc;
```

JsonDocument is the cornerstone of ArduinoJson. It’s a data structure that represents a JSON document in memory.
Now that we have the input and the JsonDocument, we can parse the input with deserializeJson():

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

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

- **DeserializationError::Ok**: the deserialization was successful.
- **DeserializationError::EmptyInput**: the input was empty or contained only spaces.
- **DeserializationError::IncompleteInput**: the input was valid but ended prematurely.
- **DeserializationError::InvalidInput**: the input was not a valid JSON document.
- **DeserializationError::NoMemory**: the JsonDocument was too small.
- **DeserializationError::TooDeep**: the input was valid, but it contained too many nesting levels; we’ll talk about that later in the book.

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

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

Second, DeserializationError has a `c_str()` member function that returns a string representation of the error. It also has an `f_str()` member that returns a Flash string, saving some space on Harvard architectures like ESP8266.

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

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

In the “Troubleshooting” chapter, we’ll look at each error code and see what can cause the error.
3.3 Extracting values from an object

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

### 3.3.1 Extracting values

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

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

This syntax leverages two C++ features:

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

### 3.3.2 Explicit casts

Some programmers avoid implicit casts because they mess with overload resolution, template parameter type deduction, and the `auto` keyword. That’s why ArduinoJson offers an alternative syntax with explicit type conversion.

#### The auto keyword

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

Here is the same code adapted for this school of thought:
**Chapter 3 Deserialze with ArduinoJson**

**3.3.3 When values are missing**

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

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

<table>
<thead>
<tr>
<th>Requested type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char*</td>
<td>nullptr</td>
</tr>
<tr>
<td>float, double</td>
<td>0.0</td>
</tr>
<tr>
<td>int, long…</td>
<td>0</td>
</tr>
<tr>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>JsonArray</td>
<td>a null object</td>
</tr>
<tr>
<td>JsonObject</td>
<td>a null object</td>
</tr>
</tbody>
</table>

The two last lines (JsonArray and JsonObject) happen when you extract a nested array or object; we’ll see that in a later section.

**No exceptions**

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

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

Here is an example:

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

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

Here is an example:

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

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

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

We’ll see a complete example that uses this syntax in the case studies.
3.4 Inspecting an unknown object

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

3.4.1 Getting a reference to the object

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

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

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

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

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

JsonObject has reference semantics

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

In a sense, we can say that JsonObject is a smart pointer. It wraps a pointer with a class that is easy to use. However, unlike the other smart pointers we talked about in the previous chapter, JsonObject doesn’t release the memory for the object when it goes out of scope because that’s the role of the JsonDocument.
3.4.2 Enumerating the keys

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

We can enumerate all pairs with a simple for loop:

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

Notice these three points about this code:

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

3.4.3 Detecting the type of value

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

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

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

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

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

### 3.4.4 Variant types and C++ types

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

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

<table>
<thead>
<tr>
<th>Variant type</th>
<th>Matching C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td><code>bool</code></td>
</tr>
<tr>
<td>Integer</td>
<td><code>char, int, long, long long, short (all signed and unsigned)</code></td>
</tr>
<tr>
<td>Float</td>
<td><code>float, double</code></td>
</tr>
<tr>
<td>String</td>
<td><code>const char*, String, std::string, std::string_view</code></td>
</tr>
<tr>
<td>Array</td>
<td><code>JsonArray, JsonArrayConst</code></td>
</tr>
<tr>
<td>Object</td>
<td><code>JsonObject, JsonObjectConst</code></td>
</tr>
</tbody>
</table>

**No naked char**

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

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

### 3.4.5 Testing if a key exists in an object

If you have an object and want to know whether a key is present, you can call `containsKey()`.

Here is an example:
// Is there a value named "text" in the object?
if (obj.containsKey("text")) {
    // Yes!
}

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

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

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

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

// Get the text of the error
const char* error = obj["error"];  

// Is there an error after all?
if (error != nullptr) {
    // ...
}

This code is faster and smaller because it only looks for the key "error" once, whereas the previous code did it twice.
3.5 Deserializing an array

3.5.1 The JSON document

We’ve seen how to parse a JSON object from GitHub’s response; it’s time to move up a notch by parsing an array of objects. Indeed, our goal is to display the top 10 of your repositories, so there will be up to 10 objects in the response. In this section, I’ll pretend there are only two repositories, so it takes less space in the book.

Here is the new sample JSON document:

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

3.5.2 Parsing the array

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

1. Put the JSON document in memory.
2. Create the JsonDocument.
3. Call deserializeJson().

Let's do it:

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

// Create the JsonDocument
JsonDocument doc;

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

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

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

3.6.1 Retrieving elements by index

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

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

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

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

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

3.6.2 Alternative syntaxes

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

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

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

All of this should sound very familiar because we’ve seen the same for objects.
3.6.3 When complex values are missing

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

Let’s see what happens in our case:

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

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

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

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

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

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

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

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

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

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

The same principles apply to null JsonArray, JsonVariant, and JsonDocument.
The null object design pattern

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

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

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

3.7 Inspecting an unknown array

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

3.7.1 Getting a reference to the array

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

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

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

Again, `JsonArray` is a reference to an array stored in the `JsonDocument`; it’s not a copy of the array. When you apply changes to the `JsonArray`, they affect the `JsonDocument` too.

3.7.2 Number of elements in an array

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

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

As the name may be confusing, let me clarify: `JsonArray::size()` returns the number of elements, not the memory consumption.
3.7.3 Iteration

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

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

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

That’s you must avoid `arr[i]` and `arr.size()` in a loop. Instead, you should use the iteration feature of `JsonArray`, like so:

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

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

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

3.7.4 Detecting the type of an element

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

Here is an example:

```cpp
// Is the first element an integer?
if (arr[0].is<int>()) {
```
// Yes!
int value = arr[0];

// ...

// Same in a loop
for (JsonVariant elem : arr) {
    // Is the current element an object?
    if (elem.is<JsonObject>()) {
        JsonObject obj = elem;
        // ...
    }
}

There is nothing new here, as it’s exactly what we saw for object members.
### 3.8 Reading from a stream

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

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

<table>
<thead>
<tr>
<th>Library</th>
<th>Class</th>
<th>Well known instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>HardwareSerial</td>
<td>Serial, Serial1...</td>
</tr>
<tr>
<td>ESP</td>
<td>BluetoothSerial</td>
<td>SerialBT</td>
</tr>
<tr>
<td></td>
<td>File</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WiFiClient</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WiFiClientSecure</td>
<td></td>
</tr>
<tr>
<td>Ethernet</td>
<td>EthernetClient</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WiFiClientUDP</td>
<td></td>
</tr>
<tr>
<td>GSM</td>
<td>GSMClient</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>File</td>
<td></td>
</tr>
<tr>
<td>SoftwareSerial</td>
<td>SoftwareSerial</td>
<td></td>
</tr>
<tr>
<td>WiFi</td>
<td>WiFiClient</td>
<td></td>
</tr>
<tr>
<td>Wire</td>
<td>TwoWire</td>
<td>Wire</td>
</tr>
</tbody>
</table>

**std::istream**

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

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

#### 3.8.1 Reading from a file

As an example, we’ll create a program that reads a JSON file stored on an SD card. We suppose this file contains the array we used as an example earlier.

The program will just read the file and print the information for each repository.
Chapter 3 Deserialize with ArduinoJson

Here is the relevant part of the code:

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

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

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

Remarks:

1. I used the .txt extension instead of .json because the FAT file system is limited to three characters for the file extension.

2. I called JsonVariant::as<T>() to pick the right overload of Serial.println().

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

You can apply the same technique to read a file in SPIFFS or LittleFS, as we'll see in the case studies.

3.8.2 Reading from an HTTP response

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

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

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

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

You can see each step in the picture below:
GitHub won’t show the token again, so don’t waste any second and write it in the source code:

```c
#define GITHUB_TOKEN "ghp_Szp27XX5wr1sZYKnBK20s0e2z2Ud0U3Lm6Iy"
```

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

```
Authorization: bearer ghp_Szp27XX5wr1sZYKnBK20s0e2z2Ud0U3Lm6Iy
```

**Certificate validation**

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

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

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

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

**The request**

To interact with the new GraphQL API, we must send a `POST` request (instead of the more common `GET` request) to the URL `https://api.github.com/graphql`.

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

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

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

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

```graphql
{
   viewer {
      name
      repositories(ownerAffiliations: OWNER,
                   orderBy: {
                      direction: DESC,
                      field: STARGAZERS
                   },
                   first: 10) {
         nodes {
            name
            stargazers {
               totalCount
            }
            issues(states: OPEN) {
               totalCount
            }
         }
      }
   }
}
```

To find the correct query, I used the GraphQL API Explorer. With this tool, you can test GraphQL queries in your browser. You can find it in GitHub’s API documentation.

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

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

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

**The response**

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

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

As you see, we call `getStream()` to get the internal stream (which is the variable `client` in our case). Unfortunately, when we do that, we bypass the part of ESP8266HTTPClient that handles chunked transfer encoding. To make sure GitHub doesn’t return a chunked response, we must set the protocol to HTTP 1.0:

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

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

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

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

Extracting the values from the JSON document is a little more complicated than what we saw earlier. Indeed, the JSON array is not at the root but under `data.viewer.repositories.nodes`, as you can see below:
So, compared to what we saw earlier, the only difference is that we’ll have to walk several objects before getting the reference to the array. The following line will do:

```plaintext
JsonArray repos = doc["data"["viewer"["repositories"["nodes")];
```
The code

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

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

// Send the request
HTTPClient http;
http.begin(client, "https://api.github.com/graphql");
http.useHTTP10(true);
http.addHeader("Authorization", "bearer " GITHUB_TOKEN);
http.POST("{\"query\":\"{viewer{name,repositories(ownerAffiliations:...\")};

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

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

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

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

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

If all works well, this program should print something like this:
You can find the complete source code of this example in the GitHub folder in the zip file provided with the book. Compared to what is shown above, the source code handles the connection to the WiFi network, checks errors, and uses Flash strings when possible.
3.9 The ArduinoJson Assistant

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

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

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

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

The Assistant for ArduinoJson 7 is composed of 3 steps:

1. Configuration.
2. JSON
3. Program
3.9.1 Step 1: Configuration

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

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

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

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

The input type can be either a char pointer, a string class, or a stream. This setting affects the memory consumption and the code generated in the last step.
In the second step, you must enter the JSON document you want to deserialize. If you’re just trying out the Assistant, you can use one of the examples at the top of the page.

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

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

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

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

At the bottom of this step, you can see the “Tweaks”, which are advanced settings that affect the computation of the memory consumption. We’ll talk about them in the “Advanced Techniques” chapter.
### 3.9.3 Step 3: Program

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

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

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

- **JsonDocument** stores the memory representation of the document.
- **deserializeJson()**:
  - `deserializeJson()` parses the input and fills the `JsonDocument`
  - `deserializeJson()` returns a `DeserializationError` that you can test with `if (err)`.
  - `deserializeJson()` can read from a `const char*`, `String`, a `std::string`, `Stream`, or `std::istream`.
- **JsonArray** and **JsonObject**:
  - You can extract values directly from the `JsonDocument` as long as there is no ambiguity.
  - To solve an ambiguity, you must call `as<JsonArray>()` or `as<JsonObject>()`.
  - `JsonArray` and `JsonObject` are references, not copies.
  - The `JsonDocument` must remain in memory; otherwise, the `JsonArray` or the `JsonObject` contains a dangling pointer.
- **JsonVariant**:
  - `JsonVariant` is also a reference and supports several types: object, array, integer, float, and boolean.
  - `JsonVariant` differs from `JsonDocument` because it doesn’t own the memory but points to it.
  - `JsonVariant` supports implicit conversion, but you can also call `as<T>()`.
- **The ArduinoJson Assistant** is an online tool that:
  - Computes the memory consumption of your program.
  - Checks that the memory consumption is within the limits of your board.
  - Generates the code to deserialize a JSON document.
In the next chapter, we’ll see how to serialize a JSON document with ArduinoJson.
That was a free chapter from “Mastering ArduinoJson”; the book contains seven chapters like this one. Here is what readers say:

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

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

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

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

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

Ready to jump in?
Go to arduinojson.org/book and use the coupon code THIRTY to get a 30% discount.