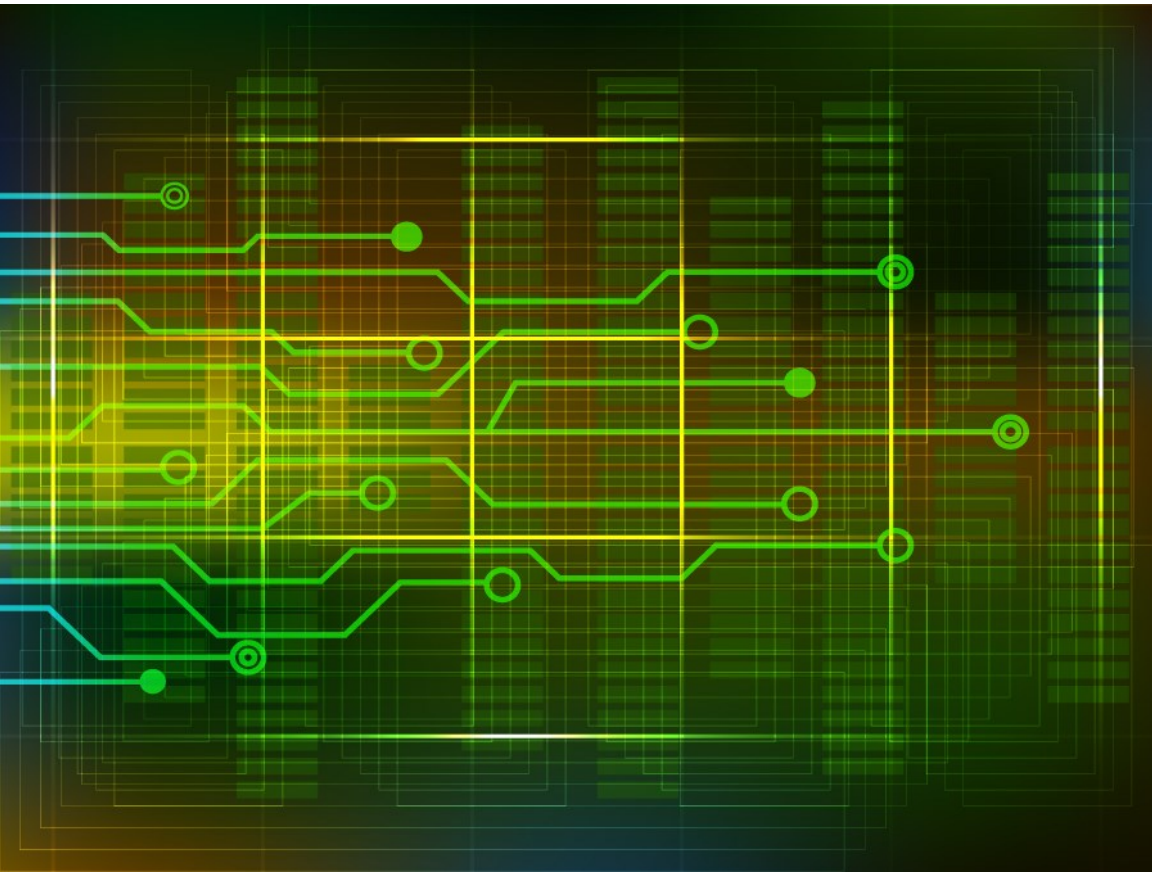
BENOIT BLANCHON
CREATOR OF ARDUINOJSON



Mastering ArduinoJson

Efficient JSON serialization for embedded C++

ArduinoJson

Contents

Chapter 3

Deserialize with ArduinoJson

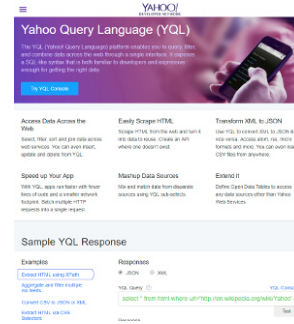
It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!

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

3.1 The example of this chapter

Now that you're familiar with JSON and C++, we're going 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.

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.

3.2.1 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 highest 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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. The macro to compute the size of an object is `JSON_OBJECT_SIZE()`. 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.

3.2.6 `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.

3.2.7 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.

3.2.8 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 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.

3.3.1 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.

3.3.2 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*`.

3.3.3 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.

3.3.4 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.

3.3.5 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];
strlcpy(hostname, config["hostname"] | "arduinojson.org", 32);
```

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

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

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.

3.4.1 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:

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

3.4.2 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 `JsonVariant`s 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.

3.4.3 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.

3.4.4 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

3.5.1 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.

3.5.2 Parse the array

You should now be familiar with the process:

1. Put the JSON document in memory.

2. Compute the size with ~~JSON_OBJECT_SIZE()~~JSON_ARRAY_SIZE().
3. Allocate the JsonBuffer.
4. Call ~~parseObject()~~parseArray().
5. Check the return value of success().

Let's do it:

```
// Put the JSON input in memory (shortened)
char input[] = "[{\"item\":{\"forecast\":{\"date\":\"09 Nov 201...";

// Compute the required size
const int capacity = JSON_ARRAY_SIZE(3)
                   + 6*JSON_OBJECT_SIZE(1)
                   + 3*JSON_OBJECT_SIZE(4);

// Allocate the JsonBuffer
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:

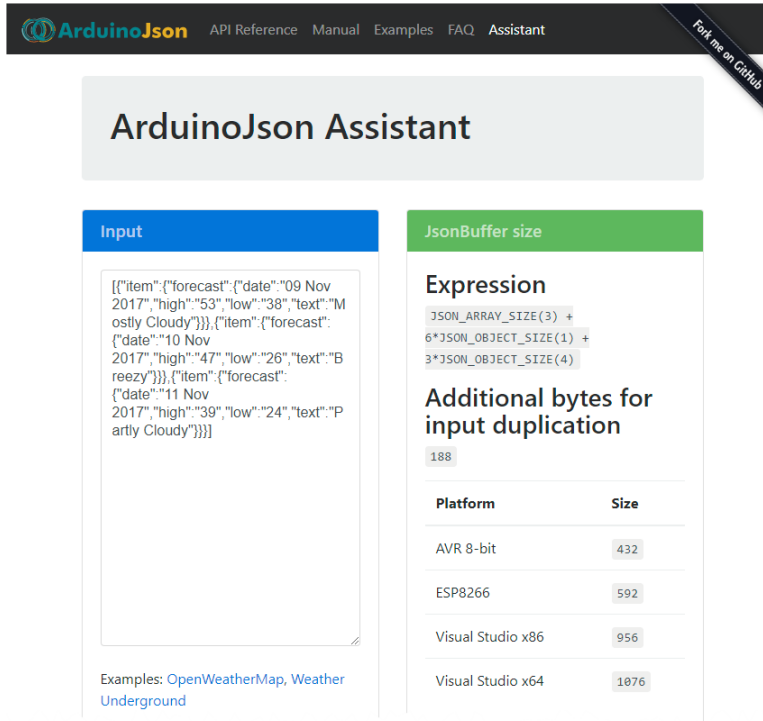
- There is one array of three elements: JSON_ARRAY_SIZE(3)
- In this array, there are three objects of one element: 3*JSON_OBJECT_SIZE(1)

- 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)`

3.5.3 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.



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

3.6.1 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"];
```

3.6.2 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&>();
```

So 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.

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`). 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 for 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.

3.7.1 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.

3.7.2 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.

3.7.3 Iteration

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

```
// BAD EXAMPLE, see below
for (int i=0; i<arr.size(); i++) {
    JsonObject& 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, to 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:

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

3.7.4 Detect the type of the elements

To 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

3.8.1 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 inserts null-terminators at the end of each string; it is the reason why this mode requires the input to be writable.

The jsmn library

As we said at the beginning of the book, jsmn is a C library that detects the tokens in the input. The zero-copy mode is very similar to the behavior of jsmn. This information should not be a surprise because the first version of ArduinoJson was just a C++ wrapper on top of jsmn.

3.8.2 An example

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

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

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

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

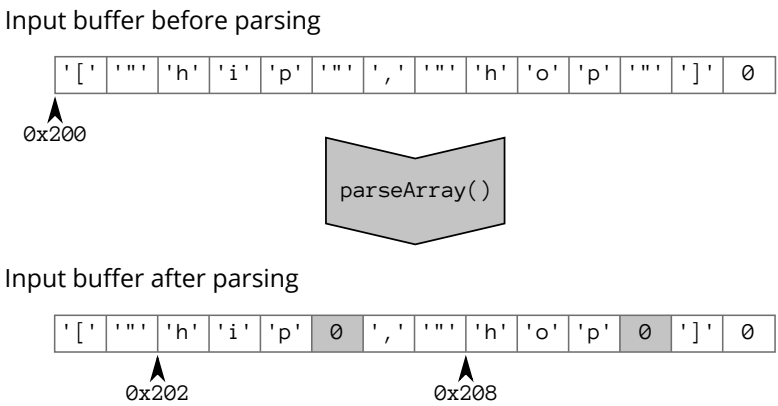
After parsing the input, when the program requests the value of the first element, ArduinoJson returns a pointer whose address is `0x202` which is the location of the string in the input buffer:

```
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.



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

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

3.8.3 Input buffer must stay in memory

As we saw, in the zero-copy mode, ArduinoJson returns pointers 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

3.9.1 The example

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

Let's go back to out 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.

3.9.2 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.

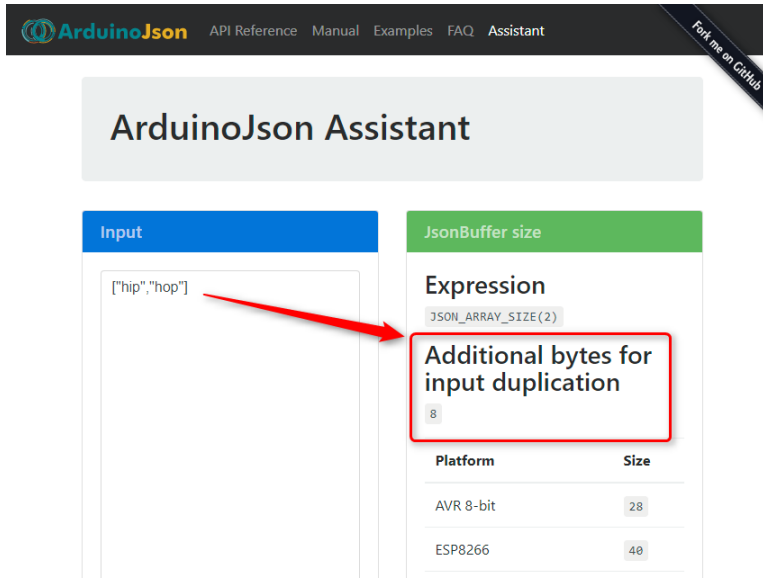
The exact capacity required is:

```
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."



3.9.3 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.

3.9.4 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[] 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	`HardwareSerial`	`Serial`, `Serial1`…
ESP8266 FS	`File`	
Ethernet	`EthernetClient`	
Ethernet	`EthernetUDP`	
GSM	`GSMClient`	
SD	`File`	
SoftwareSerial	`SoftwareSerial`	
Wifi	`WifiClient`	
Wire	`TwoWire`	`Wire`

`std::istream`

In the C++ Standard Library, an input stream is represented by the class `std::istream`.
ArduinoJson can use both `Stream` and `std::istream`.

3.10.1 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.

3.10.2 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 the (shortened) URL we need to fetch:

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

The HTTP request we need to send is:

```
GET http://query.yahooapis.com/v1/public/yql...&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":...
```

The JSON document in the body looks like that:

```
{
  "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.da...");
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.

Continue reading...

That was a free chapter from the book Mastering ArduinoJson; I hope you like it. If so, you may be interested in buying the complete text.

Not only you'll learn new skills, but you'll **contribute to sustainable open-source software**. By providing a small source of revenue to the author, you create the incentive that keeps the development running. With this model, you ensure that the building blocks of your projects stay relevant, so you don't have to switch technologies after a few years.

arduinojson.org/book

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

Thank you for your support!
Benoît