simdjson

0.4.1

Generated by Doxygen 1.8.17

Chapter 1

The Basics

An overview of what you need to know to use simdjson, with examples.

1.1 Requirements

- A recent compiler (LLVM clang6 or better, GNU GCC 7 or better) on a 64-bit (ARM or x64 Intel/AMD) POSIX systems such as macOS, freeBSD or Linux. We require that the compiler supports the C++11 standard or better.
- Visual Studio 2017 or better under 64-bit Windows. Users should target a 64-bit build (x64) instead of a 32-bit build (x86). We support the LLVM clang compiler under Visual Studio (clangcl) as well as as the regular Visual Studio compiler.

1.2 Including simdjson

To include simdjson, copy the simdjson.h and simdjson.cpp files from the singleheader directory into your project. Then include the header file in your project with:

```
#include "simdjson.h"
using namespace simdjson; // optional
```

You can compile with:

```
c++ myproject.cpp simdjson.cpp
```

Note:

- Users on macOS and other platforms were default compilers do not provide C++11 compliant by default should request it with the appropriate flag (e.g., `c++ myproject.cpp simdjson.cpp`).
- Visual Studio users should compile with the `_CRT_SECURE_NO_WARNINGS` flag to avoid warnings with respect to our use of standard C functions such as `fopen`.

1.3 Using simdjson as a CMake dependency

You can include the simdjson repository as a folder in your CMake project. In the parent `CMakeLists.txt`, include the following lines:

```
set(SIMDJSON_JUST_LIBRARY ON CACHE STRING "Build just the library, nothing else." FORCE)
add_subdirectory(simdjson EXCLUDE_FROM_ALL)
```

Elsewhere in your project, you can declare dependencies on simdjson with lines such as these:

```
add_executable(myprogram myprogram.cpp)
target_link_libraries(myprogram simdjson)
```

See `our CMake demonstration`.

1.4 The Basics: Loading and Parsing JSON Documents

The simdjson library offers a simple DOM tree API, which you can access by creating a `dom::parser` and calling the `load()` method:

```
dom::parser parser;
dom::element doc = parser.load(filename); // load and parse a file
```

Or by creating a padded string (for efficiency reasons, simdjson requires a string with SIMDJSON_PADDING bytes at the end) and calling `parse()`:

```
dom::parser parser;
dom::element doc = parser.parse("[1,2,3]"_padded); // parse a string
```

The parsed document resulting from the `parser.load` and `parser.parse` calls depends on the `parser` instance. Thus the `parser` instance must remain in scope. Furthermore, you must have at most one parsed document in play per `parser` instance.

During the`load` or `parse` calls, neither the input file nor the input string are ever modified. After calling `load` or `parse`, the source (either a file or a string) can be safely discarded. All of the JSON data is stored in the `parser` instance. The parsed document is also immutable in simdjson: you do not modify it by accessing it.

For best performance, a `parser` instance should be reused over several files: otherwise you will needlessly reallocate memory, an expensive process. It is also possible to avoid entirely memory allocations during parsing when using simdjson.

1.5 Using the Parsed JSON

Once you have an element, you can navigate it with idiomatic C++ iterators, operators and casts.

- **Extracting Values (with exceptions):** You can cast a JSON element to a native type: `double(element)` or `double x = json_element`. This works for double, uint64_t, int64_t, bool, dom::object and dom↩ ::array. An exception is thrown if the cast is not possible.
- **Extracting Values (without expceptions):** You can use a variant usage of `get()` with error codes to avoid exceptions. You first declare the variable of the appropriate type (`double`, `uint64_t`, `int64_t`, `bool`, `dom::object` and `dom::array`) and pass it by reference to `get()` which gives you back an error code: e.g.,
  ```
  simdjson::error_code error;
  simdjson::padded_string numberstring = "1.2"_padded; // our JSON input ("1.2")
  simdjson::dom::parser parser;
  double value; // variable where we store the value to be parsed
  error = parser.parse(numberstring).get(value);
  if (error) { std::cerr « error « std::endl; return EXIT_FAILURE; }
  std::cout « "I parsed " « value « " from " « numberstring.data() « std::endl;
  ```
- **Field Access:** To get the value of the "foo" field in an object, use `object["foo"]`.
- **Array Iteration:** To iterate through an array, use `for (auto value : array) { ... }`. If you know the type of the value, you can cast it right there, too! `for (double value : array) { ... }`
- **Object Iteration:** You can iterate through an object's fields, too: `for (auto [key, value] ↩ : object)`
- **Array Index:** To get at an array value by index, use the at() method: `array.at(0)` gets the first element.

  Note that array[0] does not compile, because implementing [] gives the impression indexing is a O(1) operation, which it is not presently in simdjson. Instead, you should iterate over the elements using a for-loop, as in our examples.

- **Array and Object size** Given an array or an object, you can get its size (number of elements or keys) with the `size()` method.
- **Checking an Element Type:** You can check an element's type with `element.type()`. It returns an `element_type`.

Here are some examples of all of the above:
```
auto cars_json = R"( [
  { "make": "Toyota", "model": "Camry",  "year": 2018, "tire_pressure": [ 40.1, 39.9, 37.7, 40.4 ] },
  { "make": "Kia",    "model": "Soul",   "year": 2012, "tire_pressure": [ 30.1, 31.0, 28.6, 28.7 ] },
  { "make": "Toyota", "model": "Tercel", "year": 1999, "tire_pressure": [ 29.8, 30.0, 30.2, 30.5 ] }
] )"_padded;
dom::parser parser;
// Iterating through an array of objects
for (dom::object car : parser.parse(cars_json)) {
  // Accessing a field by name
  cout « "Make/Model: " « car["make"] « "/" « car["model"] « endl;
  // Casting a JSON element to an integer
  uint64_t year = car["year"];
  cout « "- This car is " « 2020 - year « "years old." « endl;
  // Iterating through an array of floats
  double total_tire_pressure = 0;
  for (double tire_pressure : car["tire_pressure"]) {
    total_tire_pressure += tire_pressure;
  }
  cout « "- Average tire pressure: " « (total_tire_pressure / 4) « endl;
  // Writing out all the information about the car
  for (auto field : car) {
    cout « "- " « field.key « ": " « field.value « endl;
  }
}
```

Here is a different example illustrating the same ideas:
```
auto abstract_json = R"( [
    {  "12345" : {"a":12.34, "b":56.78, "c": 9998877}   },
    {  "12545" : {"a":11.44, "b":12.78, "c": 11111111}  }
  ] )"_padded;
dom::parser parser;
// Parse and iterate through an array of objects
for (dom::object obj : parser.parse(abstract_json)) {
    for(const auto& key_value : obj) {
      cout « "key: " « key_value.key « " : ";
      dom::object innerobj = key_value.value;
      cout « "a: " « double(innerobj["a"]) « ", ";
      cout « "b: " « double(innerobj["b"]) « ", ";
      cout « "c: " « int64_t(innerobj["c"]) « endl;
    }
}
```

And another one:
```
auto abstract_json = R"(
  {  "str" : { "123" : {"abc" : 3.14 } } } )"_padded;
dom::parser parser;
double v = parser.parse(abstract_json)["str"]["123"]["abc"];
cout « "number: " « v « endl;
```

1.6 C++11 Support and string_view

The simdjson library builds on compilers supporting the `C++11 standard`. It is also a strict requirement: we have no plan to support older C++ compilers.

We represent parsed strings in simdjson using the `std::string_view` class. It avoids the need to copy the data, as would be necessary with the `std::string` class. It also avoids the pitfalls of null-terminated C strings.

The `std::string_view` class has become standard as part of C++17 but it is not always available on compilers which only supports C++11. When we detect that `string_view` is natively available, we define the macro `SI↩ MDJSON_HAS_STRING_VIEW`.

When we detect that it is unavailable, we use `string-view-lite` as a substitute. In such cases, we use the type alias `using string_view = nonstd::string_view;` to
offer the same API, irrespective of the compiler and standard library. The macro `SIMDJSON_HAS_STRING_V↩ IEW` will be *undefined* to indicate that we emulate `string_view`.

1.7 C++17 Support

While the simdjson library can be used in any project using C++ 11 and above, field iteration has special support C++ 17's destructuring syntax. For example:

```
padded_string json = R"(  { "foo": 1, "bar": 2 }  )"_padded;
dom::parser parser;
dom::object object;
auto error = parser.parse(json).get(object);
if (error) { cerr « error « endl; return; }
for (auto [key, value] : object) {
  cout « key « " = " « value « endl;
}
```

For comparison, here is the C++ 11 version of the same code:

```
// C++ 11 version for comparison
padded_string json = R"(  { "foo": 1, "bar": 2 }  )"_padded;
dom::parser parser;
dom::object object;
auto error = parser.parse(json).get(object);
if (!error) { cerr « error « endl; return; }
for (dom::key_value_pair field : object) {
  cout « field.key « " = " « field.value « endl;
}
```

1.8 Minifying JSON strings without parsing

In some cases, you may have valid JSON strings that you do not wish to parse but that you wish to minify. That is, you wish to remove all unnecessary spaces. We have a fast function for this purpose (`minify`). This function does not validate your content, and it does not parse it. Instead, it assumes that your string is valid UTF-8. It is much faster than parsing the string and re-serializing it in minified form. Usage is relatively simple. You must pass an input pointer with a length parameter, as well as an output pointer and an output length parameter (by reference). The output length parameter is not read, but written to. The output pointer should point to a valid memory region that is slightly overallocated (by `simdjson::SIMDJSON_PADDING`) compared to the original string length. The input pointer and input length are read, but not written to.

```
// Starts with a valid JSON document as a string.
// It does not have to be null-terminated.
const char * some_string = "[ 1, 2, 3, 4] ";
size_t length = strlen(some_string);
// Create a buffer to receive the minified string. Make sure that there is enough room,
// including some padding (simdjson::SIMDJSON_PADDING).
std::unique_ptr<char[]> buffer{new(std::nothrow) char[length + simdjson::SIMDJSON_PADDING]};
size_t new_length{}; // It will receive the minified length.
auto error = simdjson::minify(some_string, length, buffer.get(), new_length);
// The buffer variable now has "[1,2,3,4]" and new_length has value 9.
```

Though it does not validate the JSON input, it will detect when the document ends with an unterminated string. E.g., it would refuse to minify the string `"this string is not terminated` because of the missing final quote.

1.9 UTF-8 validation (alone)

The simdjson library has fast functions to validate UTF-8 strings. They are many times faster than most functions commonly found in libraries. You can use our fast functions, even if you do not care about JSON.

```
const char * some_string = "[ 1, 2, 3, 4] ";
size_t length = strlen(some_string);
bool is_ok = simdjson::validate_utf8(some_string, length);
```

The UTF-8 validation function merely checks that the input is valid UTF-8: it works with strings in general, not just JSON strings.

Your input string does not need any padding. Any string will do. The `validate_utf8` function does not do any memory allocation on the heap, and it does not throw exceptions.

1.10 JSON Pointer

The simdjson library also supports `JSON pointer` through the at() method, letting you reach further down into the document in a single call:

```
auto cars_json = R"( [
  { "make": "Toyota", "model": "Camry",  "year": 2018, "tire_pressure": [ 40.1, 39.9, 37.7, 40.4 ] },
  { "make": "Kia",    "model": "Soul",   "year": 2012, "tire_pressure": [ 30.1, 31.0, 28.6, 28.7 ] },
  { "make": "Toyota", "model": "Tercel", "year": 1999, "tire_pressure": [ 29.8, 30.0, 30.2, 30.5 ] }
] )"_padded;
dom::parser parser;
dom::element cars = parser.parse(cars_json);
cout « cars.at("0/tire_pressure/1") « endl; // Prints 39.9
```

1.11 Error Handling

All simdjson APIs that can fail return `simdjson_result<T>`, which is a <value, error_code> pair. You can retrieve the value with .get(), like so:

```
dom::element doc;
auto error = parser.parse(json).get(doc);
if (error) { cerr « error « endl; exit(1); }
```

When you use the code this way, it is your responsibility to check for error before using the result: if there is an error, the result value will not be valid and using it will caused undefined behavior.

We can write a "quick start" example where we attempt to parse a file and access some data, without triggering exceptions:

```
#include "simdjson.h"
int main(void) {
  simdjson::dom::parser parser;
  simdjson::dom::element tweets;
  auto error = parser.load("twitter.json").get(tweets);
  if (error) { std::cerr « error « std::endl; return EXIT_FAILURE; }
  simdjson::dom::element res;
  if ((error = tweets["search_metadata"]["count"].get(res))) {
    std::cerr « "could not access keys" « std::endl;
    return EXIT_FAILURE;
  }
  std::cout « res « " results." « std::endl;
}
```

1.11.1 Error Handling Example

This is how the example in "Using the Parsed JSON" could be written using only error code checking:

```
auto cars_json = R"( [
  { "make": "Toyota", "model": "Camry",  "year": 2018, "tire_pressure": [ 40.1, 39.9, 37.7, 40.4 ] },
  { "make": "Kia",    "model": "Soul",   "year": 2012, "tire_pressure": [ 30.1, 31.0, 28.6, 28.7 ] },
  { "make": "Toyota", "model": "Tercel", "year": 1999, "tire_pressure": [ 29.8, 30.0, 30.2, 30.5 ] }
] )"_padded;
dom::parser parser;
dom::array cars;
auto error = parser.parse(cars_json).get(cars);
if (error) { cerr « error « endl; exit(1); }
// Iterating through an array of objects
for (dom::element car_element : cars) {
    dom::object car;
    if ((error = car_element.get(car))) { cerr « error « endl; exit(1); }
    // Accessing a field by name
    std::string_view make, model;
    if ((error = car["make"].get(make))) { cerr « error « endl; exit(1); }
    if ((error = car["model"].get(model))) { cerr « error « endl; exit(1); }
    cout « "Make/Model: " « make « "/" « model « endl;
    // Casting a JSON element to an integer
    uint64_t year;
    if ((error = car["year"].get(year))) { cerr « error « endl; exit(1); }
    cout « "- This car is " « 2020 - year « "years old." « endl;
    // Iterating through an array of floats
    double total_tire_pressure = 0;
    dom::array tire_pressure_array;
    if ((error = car["tire_pressure"].get(tire_pressure_array))) { cerr « error « endl; exit(1); }
    for (dom::element tire_pressure_element : tire_pressure_array) {
        double tire_pressure;
        if ((error = tire_pressure_element.get(tire_pressure))) { cerr « error « endl; exit(1); }
        total_tire_pressure += tire_pressure;
    }
    cout « "- Average tire pressure: " « (total_tire_pressure / 4) « endl;
    // Writing out all the information about the car
    for (auto field : car) {
        cout « "- " « field.key « ": " « field.value « endl;
    }
}
```

Here is another example:

```
auto abstract_json = R"( [
    {  "12345" : {"a":12.34, "b":56.78, "c": 9998877}   },
    {  "12545" : {"a":11.44, "b":12.78, "c": 11111111}  }
  ] )"_padded;
dom::parser parser;
dom::array array;
auto error = parser.parse(abstract_json).get(array);
if (error) { cerr « error « endl; exit(1); }
// Iterate through an array of objects
for (dom::element elem : array) {
    dom::object obj;
    if ((error = elem.get(obj))) { cerr « error « endl; exit(1); }
    for (auto & key_value : obj) {
        cout « "key: " « key_value.key « " : ";
        dom::object innerobj;
        if ((error = key_value.value.get(innerobj))) { cerr « error « endl; exit(1); }
        double va, vb;
        if ((error = innerobj["a"].get(va))) { cerr « error « endl; exit(1); }
        cout « "a: " « va « ", ";
        if ((error = innerobj["b"].get(vc))) { cerr « error « endl; exit(1); }
        cout « "b: " « vb « ", ";
        int64_t vc;
        if ((error = innerobj["c"].get(vc))) { cerr « error « endl; exit(1); }
        cout « "c: " « vc « endl;
    }
}
```

And another one:

```
auto abstract_json = R"(
  {  "str" : { "123" : {"abc" : 3.14 } } } )"_padded;
dom::parser parser;
double v;
auto error = parser.parse(abstract_json)["str"]["123"]["abc"].get(v);
if (error) { cerr « error « endl; exit(1); }
cout « "number: " « v « endl;
```

Notice how we can string several operations (`parser.parse(abstract_json)["str"]["123"]["abc"].get(v)`) and only check for the error once, a strategy we call *error chaining*.

The next two functions will take as input a JSON document containing an array with a single element, either a string or a number. They return true upon success.

```
simdjson::dom::parser parser{};
bool parse_double(const char *j, double &d) {
  auto error = parser.parse(j, std::strlen(j))
        .at(0)
        .get(d, error);
  if (error) { return false; }
  return true;
}
bool parse_string(const char *j, std::string &s) {
  std::string_view answer;
  auto error = parser.parse(j,strlen(j))
        .at(0)
        .get(answer, error);
  if (error) { return false; }
  s.assign(answer.data(), answer.size());
  return true;
}
```

1.11.2 Exceptions

Users more comfortable with an exception flow may choose to directly cast the `simdjson_result<T>` to the desired type:

```
dom::element doc = parser.parse(json); // Throws an exception if there was an error!
```

When used this way, a `simdjson_error` exception will be thrown if an error occurs, preventing the program from continuing if there was an error.

1.12 Tree Walking and JSON Element Types

Sometimes you don't necessarily have a document with a known type, and are trying to generically inspect or walk over JSON elements. To do that, you can use iterators and the type() method. For example, here's a quick and dirty recursive function that verbosely prints the JSON document as JSON (∗ ignoring nuances like trailing commas and escaping strings, for brevity's sake):

```
void print_json(dom::element element) {
  switch (element.type()) {
    case dom::element_type::ARRAY:
      cout « "[";
      for (dom::element child : dom::array(element)) {
        print_json(child);
        cout « ",";
      }
      cout « "]";
      break;
    case dom::element_type::OBJECT:
      cout « "{";
      for (dom::key_value_pair field : dom::object(element)) {
        cout « "\"" « field.key « "\": ";
        print_json(field.value);
      }
      cout « "}";
      break;
    case dom::element_type::INT64:
      cout « int64_t(element) « endl;
      break;
    case dom::element_type::UINT64:
      cout « uint64_t(element) « endl;
      break;
    case dom::element_type::DOUBLE:
      cout « double(element) « endl;
      break;
    case dom::element_type::STRING:
      cout « std::string_view(element) « endl;
      break;
    case dom::element_type::BOOL:
      cout « bool(element) « endl;
      break;
    case dom::element_type::NULL_VALUE:
      cout « "null" « endl;
      break;
  }
}
void basics_treewalk_1() {
  dom::parser parser;
  print_json(parser.load("twitter.json"));
}
```

1.13 Newline-Delimited JSON (ndjson) and JSON lines

The simdjson library also support multithreaded JSON streaming through a large file containing many smaller JSON documents in either `ndjson` or `JSON lines` format. If your JSON documents all contain arrays or objects, we even support direct file concatenation without whitespace. The concatenated file has no size restrictions (including larger than 4GB), though each individual document must be no larger than 4 GB.

Here is a simple example, given "x.json" with this content:

```
{ "foo": 1 }
{ "foo": 2 }
{ "foo": 3 }
dom::parser parser;
dom::document_stream docs = parser.load_many(filename);
for (dom::element doc : docs) {
  cout « doc["foo"] « endl;
}
// Prints 1 2 3
```

In-memory ndjson strings can be parsed as well, with `parser.parse_many(string)`.

Both `load_many` and `parse_many` take an optional parameter `size_t batch_size` which defines the window processing size. It is set by default to a large value (`1000000` corresponding to 1 MB). None of your JSON documents should exceed this window size, or else you will get the error `simdjson::CAPACITY`. You cannot set this window size larger than 4 GB: you will get the error `simdjson::CAPACITY`. The smaller the window size is, the less memory the function will use. Setting the window size too small (e.g., less than 100 kB) may also impact performance negatively. Leaving it to 1 MB is expected to be a good choice, unless you have some larger documents.

1.14 Thread Safety

We built simdjson with thread safety in mind.

The simdjson library is single-threaded except for `parse_many` which may use secondary threads under its control when the library is compiled with thread support.

We recommend using one `dom::parser` object per thread in which case the library is thread-safe. It is unsafe to reuse a `dom::parser` object between different threads. The parsed results (`dom::document`, `dom↩ ::element`, `array`, `object`) depend on the `dom::parser`, etc. therefore it is also potentially unsafe to use the result of the parsing between different threads.

The CPU detection, which runs the first time parsing is attempted and switches to the fastest parser for your CPU, is transparent and thread-safe.

1.15 Backwards Compatibility

The only header file supported by simdjson is `simdjson.h`. Older versions of simdjson published a number of other include files such as `document.h` or `ParsedJson.h` alongside `simdjson.h`; these headers may be moved or removed in future versions.

Chapter 2

The Basics

An overview of what you need to know to use simdjson, with examples.

2.1 Requirements

- A recent compiler (LLVM clang6 or better, GNU GCC 7 or better) on a 64-bit (ARM or x64 Intel/AMD) POSIX systems such as macOS, freeBSD or Linux. We require that the compiler supports the C++11 standard or better.
- Visual Studio 2017 or better under 64-bit Windows. Users should target a 64-bit build (x64) instead of a 32-bit build (x86). We support the LLVM clang compiler under Visual Studio (clangcl) as well as as the regular Visual Studio compiler.

2.2 Including simdjson

To include simdjson, copy `simdjson.h` and `simdjson.cpp` into your project. Then include it in your project with:

```
 {c++}
#include "simdjson.h"
using namespace simdjson; // optional
```

You can compile with:

```
c++ myproject.cpp simdjson.cpp
```

Note:

- Users on macOS and other platforms were default compilers do not provide C++11 compliant by default should request it with the appropriate flag (e.g., `c++ myproject.cpp simdjson.cpp`).
- Visual Studio users should compile with the `_CRT_SECURE_NO_WARNINGS` flag to avoid warnings with respect to our use of standard C functions such as `fopen`.

2.3 Using simdjson as a CMake dependency

You can include the simdjson repository as a folder in your CMake project. In the parent `CMakeLists.txt`, include the following lines:

```
set(SIMDJSON_JUST_LIBRARY ON CACHE STRING "Build just the library, nothing else." FORCE)
add_subdirectory(simdjson EXCLUDE_FROM_ALL)
```

Elsewhere in your project, you can declare dependencies on simdjson with lines such as these:

```
add_executable(myprogram myprogram.cpp)
target_link_libraries(myprogram simdjson)
```

See `our CMake demonstration`.

2.4 The Basics: Loading and Parsing JSON Documents

The simdjson library offers a simple DOM tree API, which you can access by creating a `dom::parser` and calling the `load()` method:

```
 {c++}
dom::parser parser;
dom::element doc = parser.load(filename); // load and parse a file
```

Or by creating a padded string (for efficiency reasons, simdjson requires a string with SIMDJSON_PADDING bytes at the end) and calling `parse()`:

```
 {c++}
dom::parser parser;
dom::element doc = parser.parse("[1,2,3]"_padded); // parse a string
```

The parsed document resulting from the `parser.load` and `parser.parse` calls depends on the `parser` instance. Thus the `parser` instance must remain in scope. Furthermore, you must have at most one parsed document in play per `parser` instance.

During the`load` or `parse` calls, neither the input file nor the input string are ever modified. After calling `load` or `parse`, the source (either a file or a string) can be safely discarded. All of the JSON data is stored in the `parser` instance. The parsed document is also immutable in simdjson: you do not modify it by accessing it.

For best performance, a `parser` instance should be reused over several files: otherwise you will needlessly reallocate memory, an expensive process. It is also possible to avoid entirely memory allocations during parsing when using simdjson. See our performance notes for details.

2.5 Using the Parsed JSON

Once you have an element, you can navigate it with idiomatic C++ iterators, operators and casts.

- **Extracting Values (with exceptions):** You can cast a JSON element to a native type: `double(element)` or `double x = json_element`. This works for double, uint64_t, int64_t, bool, dom::object and dom↵ ::array. An exception is thrown if the cast is not possible.
- **Extracting Values (without expceptions):** You can use a variant usage of `get()` with error codes to avoid exceptions. You first declare the variable of the appropriate type (`double`, `uint64_t`, `int64_t`, `bool`, `dom::object` and `dom::array`) and pass it by reference to `get()` which gives you back an error code: e.g.,

```
{c++}
 simdjson::error_code error;
 simdjson::padded_string numberstring = "1.2"_padded; // our JSON input ("1.2")
 simdjson::dom::parser parser;
 double value; // variable where we store the value to be parsed
 error = parser.parse(numberstring).get(value);
 if (error) { std::cerr « error « std::endl; return EXIT_FAILURE; }
 std::cout « "I parsed " « value « " from " « numberstring.data() « std::endl;
```

- **Field Access:** To get the value of the "foo" field in an object, use `object["foo"]`.
- **Array Iteration:** To iterate through an array, use `for (auto value : array) { ... }`. If you know the type of the value, you can cast it right there, too! `for (double value : array) { ... }`
- **Object Iteration:** You can iterate through an object's fields, too: `for (auto [key, value] ↵ : object)`
- **Array Index:** To get at an array value by index, use the at() method: `array.at(0)` gets the first element.

  Note that array[0] does not compile, because implementing [] gives the impression indexing is a O(1) operation, which it is not presently in simdjson. Instead, you should iterate over the elements using a for-loop, as in our examples.
- **Array and Object size** Given an array or an object, you can get its size (number of elements or keys) with the `size()` method.
- **Checking an Element Type:** You can check an element's type with `element.type()`. It returns an `element_type`.

Here are some examples of all of the above:

```
 {c++}
auto cars_json = R"( [
  { "make": "Toyota", "model": "Camry",  "year": 2018, "tire_pressure": [ 40.1, 39.9, 37.7, 40.4 ] },
  { "make": "Kia",    "model": "Soul",   "year": 2012, "tire_pressure": [ 30.1, 31.0, 28.6, 28.7 ] },
  { "make": "Toyota", "model": "Tercel", "year": 1999, "tire_pressure": [ 29.8, 30.0, 30.2, 30.5 ] }
] )"_padded;
dom::parser parser;
// Iterating through an array of objects
for (dom::object car : parser.parse(cars_json)) {
  // Accessing a field by name
  cout « "Make/Model: " « car["make"] « "/" « car["model"] « endl;
  // Casting a JSON element to an integer
  uint64_t year = car["year"];
  cout « "- This car is " « 2020 - year « "years old." « endl;
  // Iterating through an array of floats
  double total_tire_pressure = 0;
  for (double tire_pressure : car["tire_pressure"]) {
    total_tire_pressure += tire_pressure;
  }
  cout « "- Average tire pressure: " « (total_tire_pressure / 4) « endl;
  // Writing out all the information about the car
  for (auto field : car) {
    cout « "- " « field.key « ": " « field.value « endl;
  }
}
```

Here is a different example illustrating the same ideas:

```
 {C++}
auto abstract_json = R"( [
    {  "12345" : {"a":12.34, "b":56.78, "c": 9998877}   },
    {  "12545" : {"a":11.44, "b":12.78, "c": 11111111}  }
  ] )"_padded;
dom::parser parser;
// Parse and iterate through an array of objects
for (dom::object obj : parser.parse(abstract_json)) {
    for(const auto& key_value : obj) {
      cout « "key: " « key_value.key « " : ";
      dom::object innerobj = key_value.value;
      cout « "a: " « double(innerobj["a"]) « ", ";
      cout « "b: " « double(innerobj["b"]) « ", ";
      cout « "c: " « int64_t(innerobj["c"]) « endl;
    }
}
```

And another one:

```
{C++}
 auto abstract_json = R"(
   {  "str" : { "123" : {"abc" : 3.14 } } } )"_padded;
 dom::parser parser;
 double v = parser.parse(abstract_json)["str"]["123"]["abc"];
 cout « "number: " « v « endl;
```

2.6 C++11 Support and string_view

The simdjson library builds on compilers supporting the `C++11 standard`. It is also a strict requirement: we have no plan to support older C++ compilers.

We represent parsed strings in simdjson using the `std::string_view` class. It avoids the need to copy the data, as would be necessary with the `std::string` class. It also avoids the pitfalls of null-terminated C strings.

The `std::string_view` class has become standard as part of C++17 but it is not always available on compilers which only supports C++11. When we detect that `string_view` is natively available, we define the macro `SI↩ MDJSON_HAS_STRING_VIEW`.

When we detect that it is unavailable, we use `string-view-lite` as a substitute. In such cases, we use the type alias `using string_view = nonstd::string_view;` to
offer the same API, irrespective of the compiler and standard library. The macro `SIMDJSON_HAS_STRING_V↩ IEW` will be *undefined* to indicate that we emulate `string_view`.

2.7 C++17 Support

While the simdjson library can be used in any project using C++ 11 and above, field iteration has special support C++ 17's destructuring syntax. For example:

```
 {c++}
padded_string json = R"(  { "foo": 1, "bar": 2 }  )"_padded;
dom::parser parser;
dom::object object;
auto error = parser.parse(json).get(object);
if (error) { cerr « error « endl; return; }
for (auto [key, value] : object) {
  cout « key « " = " « value « endl;
}
```

For comparison, here is the C++ 11 version of the same code:

```
 {c++}
// C++ 11 version for comparison
padded_string json = R"(  { "foo": 1, "bar": 2 }  )"_padded;
dom::parser parser;
dom::object object;
auto error = parser.parse(json).get(object);
if (!error) { cerr « error « endl; return; }
for (dom::key_value_pair field : object) {
  cout « field.key « " = " « field.value « endl;
}
```

2.8 Minifying JSON strings without parsing

In some cases, you may have valid JSON strings that you do not wish to parse but that you wish to minify. That is, you wish to remove all unnecessary spaces. We have a fast function for this purpose (`minify`). This function does not validate your content, and it does not parse it. Instead, it assumes that your string is valid UTF-8. It is much faster than parsing the string and re-serializing it in minified form. Usage is relatively simple. You must pass an input pointer with a length parameter, as well as an output pointer and an output length parameter (by reference). The output length parameter is not read, but written to. The output pointer should point to a valid memory region that is slightly overallocated (by `simdjson::SIMDJSON_PADDING`) compared to the original string length. The input pointer and input length are read, but not written to.

```
{C++}
 // Starts with a valid JSON document as a string.
 // It does not have to be null-terminated.
 const char * some_string = "[ 1, 2, 3, 4] ";
 size_t length = strlen(some_string);
 // Create a buffer to receive the minified string. Make sure that there is enough room,
 // including some padding (simdjson::SIMDJSON_PADDING).
 std::unique_ptr<char[]> buffer{new(std::nothrow) char[length + simdjson::SIMDJSON_PADDING]};
 size_t new_length{}; // It will receive the minified length.
 auto error = simdjson::minify(some_string, length, buffer.get(), new_length);
 // The buffer variable now has "[1,2,3,4]" and new_length has value 9.
```

Though it does not validate the JSON input, it will detect when the document ends with an unterminated string. E.g., it would refuse to minify the string `"this string is not terminated` because of the missing final quote.

2.9 UTF-8 validation (alone)

The simdjson library has fast functions to validate UTF-8 strings. They are many times faster than most functions commonly found in libraries. You can use our fast functions, even if you do not care about JSON.

```
{C++}
 const char * some_string = "[ 1, 2, 3, 4] ";
 size_t length = strlen(some_string);
 bool is_ok = simdjson::validate_utf8(some_string, length);
```

The UTF-8 validation function merely checks that the input is valid UTF-8: it works with strings in general, not just JSON strings.

Your input string does not need any padding. Any string will do. The `validate_utf8` function does not do any memory allocation on the heap, and it does not throw exceptions.

2.10 JSON Pointer

The simdjson library also supports `JSON pointer` through the at() method, letting you reach further down into the document in a single call:

```
 {c++}
auto cars_json = R"( [
  { "make": "Toyota", "model": "Camry",  "year": 2018, "tire_pressure": [ 40.1, 39.9, 37.7, 40.4 ] },
  { "make": "Kia",    "model": "Soul",   "year": 2012, "tire_pressure": [ 30.1, 31.0, 28.6, 28.7 ] },
  { "make": "Toyota", "model": "Tercel", "year": 1999, "tire_pressure": [ 29.8, 30.0, 30.2, 30.5 ] }
] )"_padded;
dom::parser parser;
dom::element cars = parser.parse(cars_json);
cout « cars.at("0/tire_pressure/1") « endl; // Prints 39.9
```

2.11 Error Handling

All simdjson APIs that can fail return `simdjson_result<T>`, which is a <value, error_code> pair. You can retrieve the value with .get(), like so:

```
{c++}
dom::element doc;
auto error = parser.parse(json).get(doc);
if (error) { cerr « error « endl; exit(1); }
```

When you use the code this way, it is your responsibility to check for error before using the result: if there is an error, the result value will not be valid and using it will caused undefined behavior.

We can write a "quick start" example where we attempt to parse a file and access some data, without triggering exceptions:

```
{C++}
#include "simdjson.h"
int main(void) {
  simdjson::dom::parser parser;
  simdjson::dom::element tweets;
  auto error = parser.load("twitter.json").get(tweets);
  if (error) { std::cerr « error « std::endl; return EXIT_FAILURE; }
  simdjson::dom::element res;
  if ((error = tweets["search_metadata"]["count"].get(res))) {
    std::cerr « "could not access keys" « std::endl;
    return EXIT_FAILURE;
  }
  std::cout « res « " results." « std::endl;
}
```

2.11.1 Error Handling Example

This is how the example in "Using the Parsed JSON" could be written using only error code checking:

```
{c++}
auto cars_json = R"( [
  { "make": "Toyota", "model": "Camry",  "year": 2018, "tire_pressure": [ 40.1, 39.9, 37.7, 40.4 ] },
  { "make": "Kia",    "model": "Soul",   "year": 2012, "tire_pressure": [ 30.1, 31.0, 28.6, 28.7 ] },
  { "make": "Toyota", "model": "Tercel", "year": 1999, "tire_pressure": [ 29.8, 30.0, 30.2, 30.5 ] }
] )"_padded;
dom::parser parser;
dom::array cars;
auto error = parser.parse(cars_json).get(cars);
if (error) { cerr « error « endl; exit(1); }
// Iterating through an array of objects
for (dom::element car_element : cars) {
    dom::object car;
    if ((error = car_element.get(car))) { cerr « error « endl; exit(1); }
    // Accessing a field by name
    std::string_view make, model;
    if ((error = car["make"].get(make))) { cerr « error « endl; exit(1); }
    if ((error = car["model"].get(model))) { cerr « error « endl; exit(1); }
    cout « "Make/Model: " « make « "/" « model « endl;
    // Casting a JSON element to an integer
    uint64_t year;
    if ((error = car["year"].get(year))) { cerr « error « endl; exit(1); }
    cout « "- This car is " « 2020 - year « "years old." « endl;
    // Iterating through an array of floats
    double total_tire_pressure = 0;
    dom::array tire_pressure_array;
    if ((error = car["tire_pressure"].get(tire_pressure_array))) { cerr « error « endl; exit(1); }
    for (dom::element tire_pressure_element : tire_pressure_array) {
        double tire_pressure;
        if ((error = tire_pressure_element.get(tire_pressure))) { cerr « error « endl; exit(1); }
        total_tire_pressure += tire_pressure;
    }
    cout « "- Average tire pressure: " « (total_tire_pressure / 4) « endl;
    // Writing out all the information about the car
    for (auto field : car) {
        cout « "- " « field.key « ": " « field.value « endl;
    }
}
```

Here is another example:

```
{C++}
auto abstract_json = R"( [
    {  "12345" : {"a":12.34, "b":56.78, "c": 9998877}   },
    {  "12545" : {"a":11.44, "b":12.78, "c": 11111111}  }
```

```
  ] )"_padded;
dom::parser parser;
dom::array array;
auto error = parser.parse(abstract_json).get(array);
if (error) { cerr « error « endl; exit(1); }
// Iterate through an array of objects
for (dom::element elem : array) {
    dom::object obj;
    if ((error = elem.get(obj))) { cerr « error « endl; exit(1); }
    for (auto & key_value : obj) {
        cout « "key: " « key_value.key « " : ";
        dom::object innerobj;
        if ((error = key_value.value.get(innerobj))) { cerr « error « endl; exit(1); }
        double va, vb;
        if ((error = innerobj["a"].get(va))) { cerr « error « endl; exit(1); }
        cout « "a: " « va « ", ";
        if ((error = innerobj["b"].get(vc))) { cerr « error « endl; exit(1); }
        cout « "b: " « vb « ", ";
        int64_t vc;
        if ((error = innerobj["c"].get(vc))) { cerr « error « endl; exit(1); }
        cout « "c: " « vc « endl;
    }
}
```

And another one:

```
{C++}
 auto abstract_json = R"(
   {  "str" : { "123" : {"abc" : 3.14 } } } )"_padded;
 dom::parser parser;
 double v;
 auto error = parser.parse(abstract_json)["str"]["123"]["abc"].get(v);
 if (error) { cerr « error « endl; exit(1); }
 cout « "number: " « v « endl;
```

Notice how we can string several operations (`parser.parse(abstract_json)["str"]["123"]["abc"].get(v)`) and only check for the error once, a strategy we call *error chaining*.

The next two functions will take as input a JSON document containing an array with a single element, either a string or a number. They return true upon success.

```
 {C++}
simdjson::dom::parser parser{};
bool parse_double(const char *j, double &d) {
  auto error = parser.parse(j, std::strlen(j))
        .at(0)
        .get(d, error);
  if (error) { return false; }
  return true;
}
bool parse_string(const char *j, std::string &s) {
  std::string_view answer;
  auto error = parser.parse(j,strlen(j))
        .at(0)
        .get(answer, error);
  if (error) { return false; }
  s.assign(answer.data(), answer.size());
  return true;
}
```

2.11.2 Exceptions

Users more comfortable with an exception flow may choose to directly cast the `simdjson_result<T>` to the desired type:

```
 {c++}
dom::element doc = parser.parse(json); // Throws an exception if there was an error!
```

When used this way, a `simdjson_error` exception will be thrown if an error occurs, preventing the program from continuing if there was an error.

2.12 Tree Walking and JSON Element Types

Sometimes you don't necessarily have a document with a known type, and are trying to generically inspect or walk over JSON elements. To do that, you can use iterators and the type() method. For example, here's a quick and dirty recursive function that verbosely prints the JSON document as JSON (∗ ignoring nuances like trailing commas and escaping strings, for brevity's sake):

```
 {c++}
void print_json(dom::element element) {
  switch (element.type()) {
    case dom::element_type::ARRAY:
      cout « "[";
      for (dom::element child : dom::array(element)) {
        print_json(child);
        cout « ",";
      }
      cout « "]";
      break;
    case dom::element_type::OBJECT:
      cout « "{";
      for (dom::key_value_pair field : dom::object(element)) {
        cout « "\"" « field.key « "\": ";
        print_json(field.value);
      }
      cout « "}";
      break;
    case dom::element_type::INT64:
      cout « int64_t(element) « endl;
      break;
    case dom::element_type::UINT64:
      cout « uint64_t(element) « endl;
      break;
    case dom::element_type::DOUBLE:
      cout « double(element) « endl;
      break;
    case dom::element_type::STRING:
      cout « std::string_view(element) « endl;
      break;
    case dom::element_type::BOOL:
      cout « bool(element) « endl;
      break;
    case dom::element_type::NULL_VALUE:
      cout « "null" « endl;
      break;
  }
}
void basics_treewalk_1() {
  dom::parser parser;
  print_json(parser.load("twitter.json"));
}
```

2.13 Newline-Delimited JSON (ndjson) and JSON lines

The simdjson library also support multithreaded JSON streaming through a large file containing many smaller JSON documents in either `ndjson` or `JSON lines` format. If your JSON documents all contain arrays or objects, we even support direct file concatenation without whitespace. The concatenated file has no size restrictions (including larger than 4GB), though each individual document must be no larger than 4 GB.

Here is a simple example, given "x.json" with this content:

```
{ "foo": 1 }
{ "foo": 2 }
{ "foo": 3 }
 {c++}
dom::parser parser;
dom::document_stream docs = parser.load_many(filename);
for (dom::element doc : docs) {
  cout « doc["foo"] « endl;
}
// Prints 1 2 3
```

In-memory ndjson strings can be parsed as well, with `parser.parse_many(string)`.

Both `load_many` and `parse_many` take an optional parameter `size_t batch_size` which defines the window processing size. It is set by default to a large value (`1000000` corresponding to 1 MB). None of your JSON documents should exceed this window size, or else you will get the error `simdjson::CAPACITY`. You cannot set this window size larger than 4 GB: you will get the error `simdjson::CAPACITY`. The smaller the window size is, the less memory the function will use. Setting the window size too small (e.g., less than 100 kB) may also impact performance negatively. Leaving it to 1 MB is expected to be a good choice, unless you have some larger documents.

See parse_many.md for detailed information and design.

2.14 Thread Safety

We built simdjson with thread safety in mind.

The simdjson library is single-threaded except for \`parse_many\` which may use secondary threads under its control when the library is compiled with thread support.

We recommend using one `dom::parser` object per thread in which case the library is thread-safe. It is unsafe to reuse a `dom::parser` object between different threads. The parsed results (`dom::document`, `dom::element`, `array`, `object`) depend on the `dom::parser`, etc. therefore it is also potentially unsafe to use the result of the parsing between different threads.

The CPU detection, which runs the first time parsing is attempted and switches to the fastest parser for your CPU, is transparent and thread-safe.

2.15 Backwards Compatibility

The only header file supported by simdjson is `simdjson.h`. Older versions of simdjson published a number of other include files such as `document.h` or `ParsedJson.h` alongside `simdjson.h`; these headers may be moved or removed in future versions.

Chapter 3

CPU Architecture-Specific Implementations

3.1 Overview

The simdjson library takes advantage of SIMD instruction sets such as NEON, SSE and AVX to achieve much of its speed. Because these instruction sets work differently, simdjson has to compile a different version of the JSON parser for different CPU architectures, often with different algorithms to take better advantage of a given CPU!

The current implementations are:

- haswell: AVX2 (2013 Intel Haswell or later)
- westmere: SSE4.2 (2010 Westmere or later).
- arm64: 64-bit ARMv8-A NEON
- fallback: A generic implementation that runs on any 64-bit processor.

In many cases, you don't know where your compiled binary is going to run, so simdjson automatically compiles *all* the implementations into the executable. On Intel, it will include 3 implementations (haswell, westmere and fallback), and on ARM it will include 2 (arm64 and fallback).

If you know more about where you're going to run and want to save the space, you can disable any of these implementations at compile time with `-DSIMDJSON_IMPLEMENTATION_X=0` (where X is HASWELL, WESTMERE, ARM64 and FALLBACK).

The simdjson library automatically sets header flags for each implementation as it compiles; there is no need to set architecture-specific flags yourself (e.g., `-mavx2`, `/AVX2` or `-march=haswell`), and it may even break runtime dispatch and your binaries will fail to run on older processors.

3.2 Runtime CPU Detection

When you first use simdjson, it will detect the CPU you're running on, and swap over to the fastest implementation for it. This is a small, one-time cost and for many people will be paid the first time they call `parse()` or `load()`.

3.3 Inspecting the Detected Implementation

You can check what implementation is running with `active_implementation`:

```
 {c++}
cout « "simdjson v" « #SIMDJSON_VERSION « endl;
cout « "Detected the best implementation for your machine: " « simdjson::active_implementation->name();
cout « "(" « simdjson::active_implementation->description() « ")" « endl;
```

Implementation detection will happen in this case when you first call `name()`.

3.4 Querying Available Implementations

You can list all available implementations, regardless of which one was selected:

```
 {c++}
for (auto implementation : simdjson::available_implementations) {
  cout « implementation->name() « ": " « implementation->description() « endl;
}
```

And look them up by name:

```
 {c++}
cout « simdjson::available_implementations["fallback"]->description() « endl;
```

3.5 Manually Selecting the Implementation

If you're trying to do performance tests or see how different implementations of simdjson run, you can select the CPU architecture yourself:

```
 {c++}
// Use the fallback implementation, even though my machine is fast enough for anything
simdjson::active_implementation = simdjson::available_implementations["fallback"];
```

Chapter 4

parse_many

An interface providing features to work with files or streams containing multiple JSON documents. As fast and convenient as possible.

4.1 Contents

4.2 Motivations

The main motivation for this piece of software is to achieve maximum speed and offer a better quality of life in parsing files containing multiple JSON documents.

The JavaScript Object Notation (JSON) RFC7159 is a very handy serialization format. However, when serializing a large sequence of values as an array, or a possibly indeterminate-length or never- ending sequence of values, JSON becomes difficult to work with.

Consider a sequence of one million values, each possibly one kilobyte when encoded – roughly one gigabyte. It is often desirable to process such a dataset incrementally without having to first read all of it before beginning to produce results.

4.3 Performance

Here is a chart comparing the speed of the different alternatives to parse a multiline JSON. The simdjson library provides a threaded and non-threaded parse_many() implementation. As the figure below shows, if you can, use threads, but if you can't, it's still pretty fast!

4.4 How it works

4.4.1 Context

The parsing in simdjson is divided into 2 stages. First, in stage 1, we parse the document and find all the structural indexes (`{`, `}`, `]`, `[`, `,`, `"`, ...) and validate UTF8. Then, in stage 2, we go through the document again and build the tape using structural indexes found during stage 1. Although stage 1 finds the structural indexes, it has no knowledge of the structure of the document nor does it know whether it parsed a valid document, multiple documents, or even if the document is complete.

Prior to parse_many, most people who had to parse a multiline JSON file would proceed by reading the file line by line, using a utility function like `std::getline` or equivalent, and would then use the `parse` on each of those lines. From a performance point of view, this process is highly inefficient, in that it requires a lot of unnecessary memory allocation and makes use of the `getline` function, which is fundamentally slow, slower than the act of parsing with simdjson `(more on this here)`.

Unlike the popular parser RapidJson, our DOM does not require the buffer once the parsing job is completed, the DOM and the buffer are completely independent. The drawback of this architecture is that we need to allocate some additional memory to store our ParsedJson data, for every document inside a given file. Memory allocation can be slow and become a bottleneck, therefore, we want to minimize it as much as possible.

4.4.2 Design

To achieve a minimum amount of allocations, we opted for a design where we create only one parser object and therefore allocate its memory once, and then recycle it for every document in a given file. But, knowing that they often have largely varying size, we need to make sure that we allocate enough memory so that all the documents can fit. This value is what we call the batch size. As of right now, we need to manually specify a value for this batch size, it has to be at least as big as the biggest document in your file, but not too big so that it submerges the cached memory. The bigger the batch size, the fewer we need to make allocations. We found that 1MB is somewhat a sweet spot for now.

1. When the user calls `parse_many`, we return a `document_stream` which the user can iterate over to receive parsed documents.
2. We call stage 1 on the first batch_size bytes of JSON in the buffer, detecting structural indexes for all documents in that batch.
3. We call stage 2 on the indexes, reading tokens until we reach the end of a valid document (i.e. a single array, object, string, boolean, number or null).
4. Each time the user calls ++ to read the next document, we call stage 2 to parse the next document where we left off.
5. When we reach the end of the batch, we call stage 1 on the next batch, starting from the end of the last document, and go to step 3.

4.4.3 Threads

But how can we make use of threads if they are available? We found a pretty cool algorithm that allows us to quickly identify the position of the last JSON document in a given batch. Knowing exactly where the end of the batch is, we no longer need for stage 2 to finish in order to load a new batch. We already know where to start the next batch. Therefore, we can run stage 1 on the next batch concurrently while the main thread is going through stage 2. Running stage 1 in a different thread can, in best cases, remove almost entirely its cost and replaces it by the overhead of a thread, which is orders of magnitude cheaper. Ain't that awesome!

Thread support is only active if thread supported is detected in which case the macro SIMDJSON_THREADS_E↩ NABLED is set. Otherwise the library runs in single-thread mode.

A `document_stream` instance uses at most two threads: there is a main thread and a worker thread. You should expect the main thread to be fully occupied while the worker thread is partially busy (e.g., 80% of the time).

4.5 Support

Since we want to offer flexibility and not restrict ourselves to a specific file format, we support any file that contains any amount of valid JSON document, **separated by one or more character that is considered whitespace** by the JSON spec. Anything that is not whitespace will be parsed as a JSON document and could lead to failure.

Whitespace Characters:

- **Space**
- **Linefeed**
- **Carriage return**
- **Horizontal tab**
- **Nothing**

Some official formats ∗∗(non-exhaustive list)∗∗:

- `Newline-Delimited JSON (NDJSON)`
- `JSON lines (JSONL)`
- `Record separator-delimited JSON (RFC 7464)` <- Not supported by JsonStream!
- `More on Wikipedia...`

4.6 API

See `basics.md` for an overview of the API.

4.7 Use cases

From `jsonlines.org`:

- **Better than CSV** ```json ["Name", "Session", "Score", "Completed"] ["Gilbert", "2013", 24, true] ["Alexa", "2013", 29, true] ["May", "2012B", 14, false] ["Deloise", "2012A", 19, true] ``` CSV seems so easy that many programmers have written code to generate it themselves, and almost every implementation is different. Handling broken CSV files is a common and frustrating task. CSV has no standard encoding, no standard column separator and multiple character escaping standards. String is the only type supported for cell values, so some programs attempt to guess the correct types.

  JSON Lines handles tabular data cleanly and without ambiguity. Cells may use the standard JSON types.

  The biggest missing piece is an import/export filter for popular spreadsheet programs so that non-programmers can use this format.

- **Easy Nested Data** ```json {"name": "Gilbert", "wins": [["straight", "7"], ["one pair", "10"]]} {"name": "Alexa", "wins": [["two pair", "4"], ["two pair", "9"]]} {"name": "May", "wins": []} {"name": "Deloise", "wins": [["three of a kind", "5"]]} ``‘ JSON Lines’ biggest strength is in handling lots of similar nested data structures. One .jsonl file is easier to work with than a directory full of XML files.

Chapter 5

Performance Notes

simdjson strives to be at its fastest *without tuning*, and generally achieves this. However, there are still some scenarios where tuning can enhance performance.

5.1 Reusing the parser for maximum efficiency

If you're using simdjson to parse multiple documents, or in a loop, you should make a parser once and reuse it. The simdjson library will allocate and retain internal buffers between parses, keeping buffers hot in cache and keeping memory allocation and initialization to a minimum. In this manner, you can parse terabytes of JSON data without doing any new allocation.

```
 {c++}
dom::parser parser;
// This initializes buffers and a document big enough to handle this JSON.
dom::element doc = parser.parse("[ true, false ]"_padded);
cout « doc « endl;
// This reuses the existing buffers, and reuses and *overwrites* the old document
doc = parser.parse("[1, 2, 3]"_padded);
cout « doc « endl;
// This also reuses the existing buffers, and reuses and *overwrites* the old document
dom::element doc2 = parser.parse("true"_padded);
// Even if you keep the old reference around, doc and doc2 refer to the same document.
cout « doc « endl;
cout « doc2 « endl;
```

It's not just internal buffers though. The simdjson library reuses the document itself. The dom::element, dom↩
::object and dom::array instances are *references* to the internal document. You are only *borrowing* the document from simdjson, which purposely reuses and overwrites it each time you call parse. This prevent wasteful and unnecessary memory allocation in 99% of cases where JSON is just read, used, and converted to native values or thrown away.

You are only borrowing the document from the simdjson parser. Don't keep it long term!

This is key: don't keep the `document&`, `dom::element`, `dom::array`, `dom::object` or `string_view` objects you get back from the API. Convert them to C++ native values, structs and arrays that you own.

5.2 Server Loops: Long-Running Processes and Memory Capacity

The simdjson library automatically expands its memory capacity when larger documents are parsed, so that you don't unexpectedly fail. In a short process that reads a bunch of files and then exits, this works pretty flawlessly.

Server loops, though, are long-running processes that will keep the parser around forever. This means that if you encounter a really, really large document, simdjson will not resize back down. The simdjson library lets you adjust your allocation strategy to prevent your server from growing without bound:

- You can set a *max capacity* when constructing a parser:

```
{c++}
 dom::parser parser(1000*1000); // Never grow past documents > 1MB
 for (web_request request : listen()) {
   dom::element doc;
   auto error = parser.parse(request.body).get(doc);
   // If the document was above our limit, emit 413 = payload too large
   if (error == CAPACITY) { request.respond(413); continue; }
   // ...
 }
```

This parser will grow normally as it encounters larger documents, but will never pass 1MB.

- You can set a *fixed capacity* that never grows, as well, which can be excellent for predictability and reliability, since simdjson will never call malloc after startup!

```
{c++}
 dom::parser parser(0); // This parser will refuse to automatically grow capacity
 auto error = parser.allocate(1000*1000); // This allocates enough capacity to handle documents <= 1MB
 if (error) { cerr « error « endl; exit(1); }
 for (web_request request : listen()) {
   dom::element doc;
   error = parser.parse(request.body).get(doc);
   // If the document was above our limit, emit 413 = payload too large
   if (error == CAPACITY) { request.respond(413); continue; }
   // ...
 }
```

5.3 Large files and huge page support

There is a memory allocation performance cost the first time you process a large file (e.g. 100MB). Between the cost of allocation, the fact that the memory is not in cache, and the initial zeroing of memory, `on some systems, allocation runs far slower than parsing (e.g., 1.4GB/s)`. Reusing the parser mitigates this by paying the cost once, but does not eliminate it.

In large file use cases, enabling transparent huge page allocation on the OS can help a lot. We haven't found the right way to do this on Windows or OS/X, but on Linux, you can enable transparent huge page allocation with a command like:

```
echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

In general, when running benchmarks over large files, we recommend that you report performance numbers with and without huge pages if possible. Furthermore, you should amortize the parsing (e.g., by parsing several large files) to distinguish the time spent parsing from the time spent allocating memory. If you are using the `parse` benchmarking tool provided with the simdjson library, you can use the `-H` flag to omit the memory allocation cost from the benchmark results.

```
./parse largefile # includes memory allocation cost
./parse -H largefile # without memory allocation
```

5.4 Computed GOTOs

For best performance, we use a technique called "computed goto" when the compiler supports it, it is also sometimes described as "Labels as Values". Though it is not part of the C++ standard, it is supported by many major compilers and it brings measurable performance benefits that are difficult to achieve otherwise. The computed gotos are automatically disabled under Visual Studio.

If you wish to forcefully disable computed gotos, you can do so by compiling the code with `-DSIMDJSON_NO_COMPUTED_GOTO=1`. It is not recommended to disable computed gotos if your compiler supports it. In fact, you should almost never need to be concerned with computed gotos.

5.5 Number parsing

Some JSON files contain many floating-point values. It is the case with many GeoJSON files. Accurately parsing decimal strings into binary floating-point values with proper rounding is challenging. To our knowledge, it is not possible, in general, to parse streams of numbers at gigabytes per second using a single core. While using the simdjson library, it is possible that you might be limited to a few hundred megabytes per second if your JSON documents are densely packed with floating-point values.

- When possible, you should favor integer values written without a decimal point, as it simpler and faster to parse decimal integer values.
- When serializing numbers, you should not use more digits than necessary: 17 digits is all that is needed to exactly represent double-precision floating-point numbers. Using many more digits than necessary will make your files larger and slower to parse.
- When benchmarking parsing speeds, always report whether your JSON documents are made mostly of floating-point numbers when it is the case, since number parsing can then dominate the parsing time.

5.6 Visual Studio

On Intel and AMD Windows platforms, Microsoft Visual Studio enables programmers to build either 32-bit (x86) or 64-bit (x64) binaries. We urge you to always use 64-bit mode. Visual Studio 2019 should default on 64-bit builds when you have a 64-bit version of Windows, which we recommend.

We do not recommend that you compile simdjson with architecture-specific flags such as `arch:AVX2`. The simdjson library automatically selects the best execution kernel at runtime.

Recent versions of Microsoft Visual Studio on Windows provides support for the LLVM Clang compiler. You only need to install the "Clang compiler" optional component. You may also get a copy of the 64-bit LLVM CLang compiler for `Windows directly from LLVM`. The simdjson library fully supports the LLVM Clang compiler under Windows. In fact, you may get better performance out of simdjson with the LLVM Clang compiler than with the regular Visual Studio compiler.

5.7 Downclocking

You should not expect the simdjson library to cause downclocking of your recent Intel CPU cores.

On some Intel processors, using SIMD instructions in a sustained manner on the same CPU core may result in a phenomenon called downclocking whereas the processor initially runs these instructions at a slow speed before reducing the frequency of the core for a short time (milliseconds). Intel refers to these states as licenses. On some current Intel processors, it occurs under two scenarios:

- `Whenever 512-bit AVX-512 instructions are used`.
- Whenever heavy 256-bit or wider instructions are used. Heavy instructions are those involving floating point operations or integer multiplications (since these execute on the floating point unit).

The simdjson library does not currently support AVX-512 instructions and it does not make use of heavy 256-bit instructions. Thus there should be no downclocking due to simdjson on recent processors. You may still be worried about which SIMD instruction set is used by simdjson. Thankfully, you can always determine and change which architecture-specific implementation is used. Thus even if your CPU supports AVX2, you do not need to use AVX2. You are in control.

Chapter 6

Tape structure in simdjson

We parse a JSON document to a tape. A tape is an array of 64-bit values. Each node encountered in the JSON document is written to the tape using one or more 64-bit tape elements; the layout of the tape is in "document order": elements are stored as they are encountered in the JSON document.

Throughout, little endian encoding is assumed. The tape is indexed starting at 0 (the first element is at index 0).

6.1 Example

It is sometimes useful to start with an example. Consider the following JSON document:

```
{
    "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url": "http://www.example.com/image/481989943",
            "Height": 125,
            "Width": 100
        },
        "Animated": false,
        "IDs": [116, 943, 234, 38793]
    }
}
```

The following is a dump of the content of the tape, with the first number of each line representing the index of a tape element.

6.1.1 The Tape

index	element (64 bit word)
0	r // pointing to 38 (right after last node)
1	{ // pointing to next tape location 38 (first node after the scope)
2	string "Image"
3	{ // pointing to next tape location 37 (first node after the scope)
4	string "Width"
5	integer 800
7	string "Height"
8	integer 600
10	string "Title"

index	element (64 bit word)
11	string "View from 15th Floor"
12	string "Thumbnail"
13	{ // pointing to next tape location 23 (first node after the scope)
14	string "Url"
15	string "http://www.example.com/image/481989943"
16	string "Height"
17	integer 125
19	string "Width"
20	integer 100
22	} // pointing to previous tape location 13 (start of the scope)
23	string "Animated"
24	false
25	string "IDs"
26	[// pointing to next tape location 36 (first node after the scope)
27	integer 116
29	integer 943
31	integer 234
33	integer 38793
35	] // pointing to previous tape location 26 (start of the scope)
36	} // pointing to previous tape location 3 (start of the scope)
37	} // pointing to previous tape location 1 (start of the scope)
38	r // pointing to 0 (start root)

6.2 General formal of the tape elements

Most tape elements are written as ‘('c’ << 56) + x`where`'c'‘ is some ASCII character determining the type of the element (out of 't’, 'f', 'n', 'l', 'u', 'd', '"', '{', '}', '[', ']' ,'r') and where `x` is a 56-bit value called the payload. The payload is normally interpreted as an unsigned 56-bit integer. Note that 56-bit integers can be quite large.

Performance consideration: We believe that accessing the tape in regular units of 64 bits is more important for performance than saving memory.

6.3 Simple JSON values

Simple JSON nodes are represented with one tape element:

- null is represented as the 64-bit value ‘('n’ << 56)`where`'n'`is the 8-bit code point values (in ASCII) corresponding to the letter`'n'`.`
- `true is represented as the 64-bit value`('t' << 56)`.`
- `false is represented as the 64-bit value`('f' << 56)`.

6.4 Integer and Double values

Integer values are represented as two 64-bit tape elements:

- The 64-bit value '('l' << 56)followed by the 64-bit integer value literally. Integer values are assumed to be signed 64-bit values, using two's complement notation.
- The 64-bit value('u' << 56)` followed by the 64-bit integer value literally. Integer values are assumed to be unsigned 64-bit values.

Float values are represented as two 64-bit tape elements:

- The 64-bit value '('d' << 56)` followed by the 64-bit double value literally in standard IEEE 754 notation.

Performance consideration: We store numbers of the main tape because we believe that locality of reference is helpful for performance.

6.5 Root node

Each JSON document will have two special 64-bit tape elements representing a root node, one at the beginning and one at the end.

- The first 64-bit tape element contains the value '('r' << 56) + xwherexis the location on the tape of the last root element.
- The last 64-bit tape element contains the value('r' << 56)`.

All of the parsed document is located between these two 64-bit tape elements.

Hint: We can read the first tape element to determine the length of the tape.

6.6 Strings

We prefix the string data itself by a 32-bit header to be interpreted as a 32-bit integer. It indicates the length of the string. The actual string data starts at an offset of 4 bytes.

We store string values using UTF-8 encoding with null termination on a separate tape. A string value is represented on the main tape as the 64-bit tape element '('"' << 56) + xwhere the payloadx` is the location on the string tape of the null-terminated string.

6.7 Arrays

JSON arrays are represented using two 64-bit tape elements.

- The first 64-bit tape element contains the value '('[' << 56) + `x`where the payload`x`is 1 + the index of the second 64-bit tape element on the tape.
- The second 64-bit tape element contains the value(']' << 56) + `x`where the payload`x`` contains the index of the first 64-bit tape element on the tape.

All the content of the array is located between these two tape elements, including arrays and objects.

Performance consideration: We can skip the content of an array entirely by accessing the first 64-bit tape element, reading the payload and moving to the corresponding index on the tape.

6.8 Objects

JSON objects are represented using two 64-bit tape elements.

- The first 64-bit tape element contains the value '('{' << 56) + `x`where the payload`x`is 1 + the index of the second 64-bit tape element on the tape.
- The second 64-bit tape element contains the value('}' << 56) + `x`where the payload`x`` contains the index of the first 64-bit tape element on the tape.

In-between these two tape elements, we alternate between key (which must be strings) and values. A value could be an object or an array.

All the content of the object is located between these two tape elements, including arrays and objects.

Performance consideration: We can skip the content of an object entirely by accessing the first 64-bit tape element, reading the payload and moving to the corresponding index on the tape.

Chapter 7

Deprecated List

File simdjson.h

We'll be removing this file so it isn't confused with the top level simdjson.h

Chapter 8

Hierarchical Index

8.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 9

Class Index

9.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 10

File Index

10.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 11

Class Documentation

11.1 simdjson::dom::array Class Reference

JSON array.

Classes

- class iterator

Public Member Functions

- really_inline array () noexcept

    Create a new, invalid array.
- iterator begin () const noexcept

    Return the first array element.
- iterator end () const noexcept

    One past the last array element.
- size_t size () const noexcept

    Get the size of the array (number of immediate children).
- simdjson_result< element > at (const std::string_view &json_pointer) const noexcept

    Get the value associated with the given JSON pointer.
- simdjson_result< element > at (size_t index) const noexcept

    Get the value at the given index.

11.1.1 Detailed Description

JSON array.

Definition at line 19 of file array.h.

11.1.2 Member Function Documentation

11.1.2.1 at() [1/2]

```
simdjson_result< element > simdjson::dom::array::at (
            const std::string_view & json_pointer ) const  [inline], [noexcept]
```

Get the value associated with the given JSON pointer.

dom::parser parser; array a = parser.parse(R"([{ "foo": { "a": [10, 20, 30] }}])"_padded); a.at("0/foo/a/1") == 20 a.at("0")["foo"]["a"].at(1) == 20

Returns

The value associated with the given JSON pointer, or:

- NO_SUCH_FIELD if a field does not exist in an object
- INDEX_OUT_OF_BOUNDS if an array index is larger than an array length
- INCORRECT_TYPE if a non-integer is used to access an array
- INVALID_JSON_POINTER if the JSON pointer is invalid and cannot be parsed

Definition at line 64 of file array.h.

11.1.2.2 at() [2/2]

```
simdjson_result< element > simdjson::dom::array::at (
            size_t index ) const  [inline], [noexcept]
```

Get the value at the given index.

This function has linear-time complexity and is equivalent to the following:

size_t i=0; for (auto element : ∗this) { if (i == index) { return element; } i++; } return INDEX_OUT_OF_BOUNDS;

Avoid calling the at() function repeatedly.

Returns

The value at the given index, or:

- INDEX_OUT_OF_BOUNDS if the array index is larger than an array length

Definition at line 93 of file array.h.

11.1.2.3 begin()

```
array::iterator simdjson::dom::array::begin ( ) const  [inline], [noexcept]
```

Return the first array element.

Part of the std::iterable interface.

Definition at line 55 of file array.h.

11.1.2.4 end()

```
array::iterator simdjson::dom::array::end ( ) const  [inline], [noexcept]
```

One past the last array element.

Part of the std::iterable interface.

Definition at line 58 of file array.h.

11.1.2.5 size()

```
size_t simdjson::dom::array::size ( ) const  [inline], [noexcept]
```

Get the size of the array (number of immediate children).

It is a saturated value with a maximum of 0xFFFFFF: if the value is 0xFFFFFF then the size is 0xFFFFFF or greater.

Definition at line 61 of file array.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/array.h

11.2 simdjson::dom::document Class Reference

A parsed JSON document.

Public Member Functions

- document () noexcept=default

  Create a document container with zero capacity.
- document (document &&other) noexcept=default

  Take another document's buffers.
- document & operator= (document &&other) noexcept=default

  Take another document's buffers.
- element root () const noexcept

  Get the root element of this document as a JSON array.

11.2.1 Detailed Description

A parsed JSON document.

This class cannot be copied, only moved, to avoid unintended allocations.

Definition at line 19 of file document.h.

11.2.2 Constructor & Destructor Documentation

11.2.2.1 document() [1/2]

```
simdjson::dom::document::document ( )  [default], [noexcept]
```

Create a document container with zero capacity.

The parser will allocate capacity as needed.

11.2.2.2 document() [2/2]

```
simdjson::dom::document::document (
            document && other )  [default], [noexcept]
```

Take another document's buffers.

Parameters

other	The document to take. Its capacity is zeroed and it is invalidated.

11.2.3 Member Function Documentation

11.2.3.1 operator=()

```
document& simdjson::dom::document::operator= (
            document && other )  [default], [noexcept]
```

Take another document's buffers.

Parameters

other	The document to take. Its capacity is zeroed.

The documentation for this class was generated from the following file:

- include/simdjson/dom/document.h

11.3 simdjson::dom::document_stream Class Reference

A forward-only stream of documents.

Classes

- class iterator

  An iterator through a forward-only stream of documents.

Public Member Functions

- really_inline document_stream () noexcept

  Construct an uninitialized document_stream.
- really_inline document_stream (document_stream &&other) noexcept=default

  Move one document_stream to another.
- really_inline document_stream & operator= (document_stream &&other) noexcept=default

  Move one document_stream to another.
- really_inline iterator begin () noexcept

  Start iterating the documents in the stream.
- really_inline iterator end () noexcept

  The end of the stream, for iterator comparison purposes.

11.3.1 Detailed Description

A forward-only stream of documents.

Produced by parser::parse_many.

Definition at line 73 of file document_stream.h.

11.3.2 Constructor & Destructor Documentation

11.3.2.1 document_stream()

```
really_inline simdjson::dom::document_stream::document_stream ( )  [noexcept]
```

Construct an uninitialized document_stream.

```
{c++}
document_stream docs;
error = parser.parse_many(json).get(docs);
```

Definition at line 86 of file document_stream.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/document_stream.h

11.4 simdjson::dom::element Class Reference

A JSON element.

Public Member Functions

- really_inline element () noexcept

  Create a new, invalid element.
- really_inline element_type type () const noexcept

  The type of this element.
- simdjson_result< array > get_array () const noexcept

  Cast this element to an array.
- simdjson_result< object > get_object () const noexcept

  Cast this element to an object.
- simdjson_result< const char * > get_c_str () const noexcept

  Cast this element to a null-terminated C string.
- simdjson_result< size_t > get_string_length () const noexcept

  Gives the length in bytes of the string.
- simdjson_result< std::string_view > get_string () const noexcept

  Cast this element to a string.
- simdjson_result< int64_t > get_int64 () const noexcept

  Cast this element to a signed integer.
- simdjson_result< uint64_t > get_uint64 () const noexcept

  Cast this element to an unsigned integer.
- simdjson_result< double > get_double () const noexcept

  Cast this element to an double floating-point.
- simdjson_result< bool > get_bool () const noexcept

  Cast this element to a bool.
- bool is_array () const noexcept

  Whether this element is a json array.
- bool is_object () const noexcept

  Whether this element is a json object.
- bool is_string () const noexcept

  Whether this element is a json string.
- bool is_int64 () const noexcept

  Whether this element is a json number that fits in a signed 64-bit integer.
- bool is_uint64 () const noexcept

  Whether this element is a json number that fits in an unsigned 64-bit integer.
- bool is_double () const noexcept

  Whether this element is a json number that fits in a double.
- bool is_number () const noexcept

  Whether this element is a json number.
- bool is_bool () const noexcept

  Whether this element is a json `true` or `false`.
- bool is_null () const noexcept

  Whether this element is a json `null`.
- template<typename T >
  really_inline bool is () const noexcept

  Tell whether the value can be cast to provided type (T).
- template<typename T >
  simdjson_result< T > get () const noexcept

  Get the value as the provided type (T).
- template<typename T >
  WARN_UNUSED really_inline error_code get (T &value) const noexcept

  Get the value as the provided type (T).

- template<typename T >
  void tie (T &value, error_code &error) &&noexcept

  Get the value as the provided type (T), setting error if it's not the given type.
- operator bool () const noexcept(false)

  Read this element as a boolean.
- operator const char ∗ () const noexcept(false)

  Read this element as a null-terminated UTF-8 string.
- operator std::string_view () const noexcept(false)

  Read this element as a null-terminated UTF-8 string.
- operator uint64_t () const noexcept(false)

  Read this element as an unsigned integer.
- operator int64_t () const noexcept(false)

  Read this element as an signed integer.
- operator double () const noexcept(false)

  Read this element as an double.
- operator array () const noexcept(false)

  Read this element as a JSON array.
- operator object () const noexcept(false)

  Read this element as a JSON object (key/value pairs).
- dom::array::iterator begin () const noexcept(false)

  Iterate over each element in this array.
- dom::array::iterator end () const noexcept(false)

  Iterate over each element in this array.
- simdjson_result< element > operator[ ] (const std::string_view &key) const noexcept

  Get the value associated with the given key.
- simdjson_result< element > operator[ ] (const char ∗key) const noexcept

  Get the value associated with the given key.
- simdjson_result< element > at (const std::string_view &json_pointer) const noexcept

  Get the value associated with the given JSON pointer.
- simdjson_result< element > at (size_t index) const noexcept

  Get the value at the given index.
- simdjson_result< element > at_key (const std::string_view &key) const noexcept

  Get the value associated with the given key.
- simdjson_result< element > at_key_case_insensitive (const std::string_view &key) const noexcept

  Get the value associated with the given key in a case-insensitive manner.
- template<> WARN_UNUSED really_inline error_code **get** (element &value) const noexcept
- template<> simdjson_result< array > **get** () const noexcept
- template<> simdjson_result< object > **get** () const noexcept
- template<> simdjson_result< const char ∗ > **get** () const noexcept
- template<> simdjson_result< int64_t > **get** () const noexcept
- template<> simdjson_result< uint64_t > **get** () const noexcept
- template<> simdjson_result< double > **get** () const noexcept
- template<> simdjson_result< bool > **get** () const noexcept

11.4.1 Detailed Description

A JSON element.

References an element in a JSON document, representing a JSON null, boolean, string, number, array or object.

Definition at line 38 of file element.h.

11.4.2 Member Function Documentation

11.4.2.1 at() [1/2]

```
simdjson_result< element > simdjson::dom::element::at (
            const std::string_view & json_pointer ) const  [inline], [noexcept]
```

Get the value associated with the given JSON pointer.

dom::parser parser; element doc = parser.parse(R"({ "foo": { "a": [10, 20, 30] }})"_padded); doc.at("/foo/a/1") == 20 doc.at("/")["foo"]["a"].at(1) == 20 doc.at("")["foo"]["a"].at(1) == 20

Returns

The value associated with the given JSON pointer, or:

- NO_SUCH_FIELD if a field does not exist in an object
- INDEX_OUT_OF_BOUNDS if an array index is larger than an array length
- INCORRECT_TYPE if a non-integer is used to access an array
- INVALID_JSON_POINTER if the JSON pointer is invalid and cannot be parsed

Definition at line 343 of file element.h.

11.4.2.2 at() [2/2]

```
simdjson_result< element > simdjson::dom::element::at (
            size_t index ) const  [inline], [noexcept]
```

Get the value at the given index.

Returns

The value at the given index, or:

- INDEX_OUT_OF_BOUNDS if the array index is larger than an array length

Definition at line 353 of file element.h.

11.4.2.3 at_key()

```
simdjson_result< element > simdjson::dom::element::at_key (
            const std::string_view & key ) const  [inline], [noexcept]
```

Get the value associated with the given key.

The key will be matched against **unescaped** JSON:

dom::parser parser; parser.parse(R"({ "a
": 1 })"_padded)["a\n"].get<uint64_t>().first == 1 parser.parse(R"({ "a
": 1 })"_padded)["a\\n"].get<uint64_t>().error() == NO_SUCH_FIELD

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object

Definition at line 356 of file element.h.

11.4.2.4 at_key_case_insensitive()

```
simdjson_result< element > simdjson::dom::element::at_key_case_insensitive (
            const std::string_view & key ) const  [inline], [noexcept]
```

Get the value associated with the given key in a case-insensitive manner.

Note: The key will be matched against **unescaped** JSON.

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object

Definition at line 359 of file element.h.

11.4.2.5 begin()

```
array::iterator simdjson::dom::element::begin ( ) const  [inline], [noexcept]
```

Iterate over each element in this array.

Returns

The beginning of the iteration.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not an array

Definition at line 328 of file element.h.

11.4.2.6 end()

```
array::iterator simdjson::dom::element::end ( ) const  [inline], [noexcept]
```

Iterate over each element in this array.

Returns

The end of the iteration.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not an array

Definition at line 331 of file element.h.

11.4.2.7 get() [1/2]

```
template<typename T >
simdjson_result<T> simdjson::dom::element::get ( ) const  [inline], [noexcept]
```

Get the value as the provided type (T).

Supported types:

- Boolean: bool
- Number: double, uint64_t, int64_t
- String: std::string_view, const char ∗
- Array: dom::array
- Object: dom::object

Template Parameters

T	bool, double, uint64_t, int64_t, std::string_view, const char ∗, dom::array, dom::object

Returns

The value cast to the given type, or: INCORRECT_TYPE if the value cannot be cast to the given type.

11.4.2.8 get() [2/2]

```
template<typename T >
WARN_UNUSED really_inline error_code simdjson::dom::element::get (
            T & value ) const  [noexcept]
```

Get the value as the provided type (T).

Supported types:

- Boolean: bool
- Number: double, uint64_t, int64_t
- String: std::string_view, const char ∗
- Array: dom::array
- Object: dom::object

Template Parameters

T	bool, double, uint64_t, int64_t, std::string_view, const char ∗, dom::array, dom::object

Parameters

value	The variable to set to the value. May not be set if there is an error.

Returns

The error that occurred, or SUCCESS if there was no error.

Definition at line 280 of file element.h.

11.4.2.9 get_array()

```
simdjson_result< array > simdjson::dom::element::get_array ( ) const  [inline], [noexcept]
```

Cast this element to an array.

Equivalent to get<array>().

Returns

An object that can be used to iterate the array, or: INCORRECT_TYPE if the JSON element is not an array.

Definition at line 262 of file element.h.

11.4.2.10 get_bool()

```
simdjson_result< bool > simdjson::dom::element::get_bool ( ) const  [inline], [noexcept]
```

Cast this element to a bool.

Equivalent to get<bool>().

Returns

A bool value. Returns INCORRECT_TYPE if the JSON element is not a boolean.

Definition at line 180 of file element.h.

11.4.2.11 get_c_str()

```
simdjson_result< const char * > simdjson::dom::element::get_c_str ( ) const  [inline], [noexcept]
```

Cast this element to a null-terminated C string.

The string is guaranteed to be valid UTF-8.

The get_c_str() function is equivalent to get<const char *>().

The length of the string is given by get_string_length(). Because JSON strings may contain null characters, it may be incorrect to use strlen to determine the string length.

It is possible to get a single string_view instance which represents both the string content and its length: see get_string().

Returns

A pointer to a null-terminated UTF-8 string. This string is stored in the parser and will be invalidated the next time it parses a document or when it is destroyed. Returns INCORRECT_TYPE if the JSON element is not a string.

Definition at line 188 of file element.h.

11.4.2.12 get_double()

```
simdjson_result< double > simdjson::dom::element::get_double ( ) const  [inline], [noexcept]
```

Cast this element to an double floating-point.

Equivalent to get<double>().

Returns

A double value. Returns INCORRECT_TYPE if the JSON element is not a number.

Definition at line 241 of file element.h.

11.4.2.13 get_int64()

```
simdjson_result< int64_t > simdjson::dom::element::get_int64 ( ) const  [inline], [noexcept]
```

Cast this element to a signed integer.

Equivalent to get<int64_t>().

Returns

A signed 64-bit integer. Returns INCORRECT_TYPE if the JSON element is not an integer, or NUMBER_↩ OUT_OF_RANGE if it is negative.

Definition at line 227 of file element.h.

11.4.2.14 get_object()

```
simdjson_result< object > simdjson::dom::element::get_object ( ) const  [inline], [noexcept]
```

Cast this element to an object.

Equivalent to get<object>().

Returns

An object that can be used to look up or iterate the object's fields, or: INCORRECT_TYPE if the JSON element is not an object.

Definition at line 270 of file element.h.

11.4.2.15 get_string()

```
simdjson_result< std::string_view > simdjson::dom::element::get_string ( ) const  [inline],
[noexcept]
```

Cast this element to a string.

The string is guaranteed to be valid UTF-8.

Equivalent to get<std::string_view>().

Returns

An UTF-8 string. The string is stored in the parser and will be invalidated the next time it parses a document or when it is destroyed. Returns INCORRECT_TYPE if the JSON element is not a string.

Definition at line 206 of file element.h.

11.4.2.16 get_string_length()

```
simdjson_result< size_t > simdjson::dom::element::get_string_length ( ) const  [inline], [noexcept]
```

Gives the length in bytes of the string.

It is possible to get a single string_view instance which represents both the string content and its length: see get_string().

Returns

A string length in bytes. Returns INCORRECT_TYPE if the JSON element is not a string.

Definition at line 197 of file element.h.

11.4.2.17 get_uint64()

```
simdjson_result< uint64_t > simdjson::dom::element::get_uint64 ( ) const  [inline], [noexcept]
```

Cast this element to an unsigned integer.

Equivalent to get<uint64_t>().

Returns

An unsigned 64-bit integer. Returns INCORRECT_TYPE if the JSON element is not an integer, or NUMBE↩ R_OUT_OF_RANGE if it is too large.

Definition at line 214 of file element.h.

11.4.2.18 is()

```
template<typename T >
really_inline bool simdjson::dom::element::is  [noexcept]
```

Tell whether the value can be cast to provided type (T).

Supported types:

- Boolean: bool
- Number: double, uint64_t, int64_t
- String: std::string_view, const char *
- Array: dom::array
- Object: dom::object

Template Parameters

T	bool, double, uint64_t, int64_t, std::string_view, const char ∗, dom::array, dom::object

Definition at line 291 of file element.h.

11.4.2.19 is_array()

```
bool simdjson::dom::element::is_array ( ) const  [inline], [noexcept]
```

Whether this element is a json array.

Equivalent to is<array>().

Definition at line 305 of file element.h.

11.4.2.20 is_bool()

```
bool simdjson::dom::element::is_bool ( ) const  [inline], [noexcept]
```

Whether this element is a json `true` or `false`.

Equivalent to is<bool>().

Definition at line 311 of file element.h.

11.4.2.21 is_double()

```
bool simdjson::dom::element::is_double ( ) const  [inline], [noexcept]
```

Whether this element is a json number that fits in a double.

Equivalent to is<double>().

Definition at line 310 of file element.h.

11.4.2.22 is_int64()

```
bool simdjson::dom::element::is_int64 ( ) const  [inline], [noexcept]
```

Whether this element is a json number that fits in a signed 64-bit integer.

Equivalent to is<int64_t>().

Definition at line 308 of file element.h.

11.4.2.23 is_number()

```
bool simdjson::dom::element::is_number ( ) const  [inline], [noexcept]
```

Whether this element is a json number.

Both integers and floating points will return true.

11.4.2.24 is_object()

```
bool simdjson::dom::element::is_object ( ) const  [inline], [noexcept]
```

Whether this element is a json object.

Equivalent to is<object>().

Definition at line 306 of file element.h.

11.4.2.25 is_string()

```
bool simdjson::dom::element::is_string ( ) const  [inline], [noexcept]
```

Whether this element is a json string.

Equivalent to is<std::string_view>() or is<const char ∗>().

Definition at line 307 of file element.h.

11.4.2.26 is_uint64()

```
bool simdjson::dom::element::is_uint64 ( ) const  [inline], [noexcept]
```

Whether this element is a json number that fits in an unsigned 64-bit integer.

Equivalent to is<uint64_t>().

Definition at line 309 of file element.h.

11.4.2.27 operator array()

```
simdjson::dom::element::operator array ( ) const  [inline], [noexcept]
```

Read this element as a JSON array.

Returns

The JSON array.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not an array

Definition at line 325 of file element.h.

11.4.2.28 operator bool()

```
simdjson::dom::element::operator bool ( ) const  [inline], [noexcept]
```

Read this element as a boolean.

Returns

The boolean value

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not a boolean.

Definition at line 319 of file element.h.

11.4.2.29 operator const char *()

```
simdjson::dom::element::operator const char * ( ) const  [inline], [explicit], [noexcept]
```

Read this element as a null-terminated UTF-8 string.

Be mindful that JSON allows strings to contain null characters.

Does *not* convert other types to a string; requires that the JSON type of the element was an actual string.

Returns

The string value.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not a string.

Definition at line 320 of file element.h.

11.4.2.30 operator double()

```
simdjson::dom::element::operator double ( ) const  [inline], [noexcept]
```

Read this element as an double.

Returns

The double value.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not a number
simdjson_error(NUMBER_OUT_OF_RANGE)	if the integer doesn't fit in 64 bits or is negative

Definition at line 324 of file element.h.

11.4.2.31 operator int64_t()

```
simdjson::dom::element::operator int64_t ( ) const  [inline], [noexcept]
```

Read this element as an signed integer.

Returns

The integer value.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not an integer
simdjson_error(NUMBER_OUT_OF_RANGE)	if the integer doesn't fit in 64 bits

Definition at line 323 of file element.h.

11.4.2.32 operator object()

```
simdjson::dom::element::operator object ( ) const  [inline], [noexcept]
```

Read this element as a JSON object (key/value pairs).

Returns

The JSON object.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not an object

Definition at line 326 of file element.h.

11.4.2.33 operator std::string_view()

```
simdjson::dom::element::operator std::string_view ( ) const  [inline], [noexcept]
```

Read this element as a null-terminated UTF-8 string.

Does *not* convert other types to a string; requires that the JSON type of the element was an actual string.

Returns

The string value.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not a string.

Definition at line 321 of file element.h.

11.4.2.34 operator uint64_t()

```
simdjson::dom::element::operator uint64_t ( ) const  [inline], [noexcept]
```

Read this element as an unsigned integer.

Returns

The integer value.

Exceptions

simdjson_error(INCORRECT_TYPE)	if the JSON element is not an integer
simdjson_error(NUMBER_OUT_OF_RANGE)	if the integer doesn't fit in 64 bits or is negative

Definition at line 322 of file element.h.

11.4.2.35 operator[]() [1/2]

```
simdjson_result< element > simdjson::dom::element::operator[] (
            const char * key ) const  [inline], [noexcept]
```

Get the value associated with the given key.

The key will be matched against **unescaped** JSON:

dom::parser parser; parser.parse(R"({ "a
": 1 })"_padded)["a\n"].get<uint64_t>().first == 1 parser.parse(R"({ "a
": 1 })"_padded)["a\\n"].get<uint64_t>().error() == NO_SUCH_FIELD

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object
- INCORRECT_TYPE if this is not an object

Definition at line 340 of file element.h.

11.4.2.36 operator[]() [2/2]

```
simdjson_result< element > simdjson::dom::element::operator[] (
            const std::string_view & key ) const  [inline], [noexcept]
```

Get the value associated with the given key.

The key will be matched against **unescaped** JSON:

dom::parser parser; parser.parse(R"({ "a
": 1 })"_padded)["a\n"].get<uint64_t>().first == 1 parser.parse(R"({ "a
": 1 })"_padded)["a\\n"].get<uint64_t>().error() == NO_SUCH_FIELD

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object
- INCORRECT_TYPE if this is not an object

Definition at line 337 of file element.h.

11.4.2.37 tie()

```
template<typename T >
void simdjson::dom::element::tie (
            T & value,
            error_code & error ) &&  [inline], [noexcept]
```

Get the value as the provided type (T), setting error if it's not the given type.

Supported types:

- Boolean: bool
- Number: double, uint64_t, int64_t
- String: std::string_view, const char *
- Array: dom::array
- Object: dom::object

Template Parameters

T	bool, double, uint64_t, int64_t, std::string_view, const char ∗, dom::array, dom::object

Parameters

value	The variable to set to the given type. value is undefined if there is an error.
error	The variable to store the error. error is set to error_code::SUCCEED if there is an error.

The documentation for this class was generated from the following file:

- include/simdjson/dom/element.h

11.5 simdjson::implementation Class Reference

An implementation of simdjson for a particular CPU architecture.

Public Member Functions

- virtual const std::string & name () const

    The name of this implementation.
- virtual const std::string & description () const

    The description of this implementation.
- virtual WARN_UNUSED bool validate_utf8 (const char ∗buf, size_t len) const noexcept=0

    Validate the UTF-8 string.

11.5.1 Detailed Description

An implementation of simdjson for a particular CPU architecture.

Also used to maintain the currently active implementation. The active implementation is automatically initialized on first use to the most advanced implementation supported by the host.

Definition at line 52 of file implementation.h.

11.5.2 Member Function Documentation

11.5.2.1 description()

```
virtual const std::string& simdjson::implementation::description ( ) const  [inline], [virtual]
```

The description of this implementation.

```
const implementation *impl = simdjson::active_implementation;
cout << "simdjson is optimized for " << impl->name() << "(" << impl->description() << ")" << endl;
```

Returns

the name of the implementation, e.g. "haswell", "westmere", "arm64"

Definition at line 73 of file implementation.h.

11.5.2.2 name()

```
virtual const std::string& simdjson::implementation::name ( ) const  [inline], [virtual]
```

The name of this implementation.

```
const implementation *impl = simdjson::active_implementation;
cout << "simdjson is optimized for " << impl->name() << "(" << impl->description() << ")" << endl;
```

Returns

the name of the implementation, e.g. "haswell", "westmere", "arm64"

Definition at line 63 of file implementation.h.

11.5.2.3 validate_utf8()

```
virtual WARN_UNUSED bool simdjson::implementation::validate_utf8 (
             const char * buf,
             size_t len ) const  [pure virtual], [noexcept]
```

Validate the UTF-8 string.

Overridden by each implementation.

Parameters

buf	the string to validate.
len	the length of the string in bytes.

Returns

true if and only if the string is valid UTF-8.

The documentation for this class was generated from the following file:

- include/simdjson/implementation.h

11.6 simdjson::dom::array::iterator Class Reference

Public Member Functions

- element operator∗ () const noexcept

  Get the actual value.
- iterator & operator++ () noexcept

  Get the next value.
- bool operator!= (const iterator &other) const noexcept

  Check if these values come from the same place in the JSON.

11.6.1 Detailed Description

Definition at line 24 of file array.h.

11.6.2 Member Function Documentation

11.6.2.1 operator"!=()

```
bool simdjson::dom::array::iterator::operator!= (
             const iterator & other ) const  [inline], [noexcept]
```

Check if these values come from the same place in the JSON.

Part of the std::iterator interface.

Definition at line 109 of file array.h.

11.6.2.2 operator++()

```
array::iterator & simdjson::dom::array::iterator::operator++ ( )  [inline], [noexcept]
```

Get the next value.

Part of the std::iterator interface.

Definition at line 112 of file array.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/array.h

11.7 simdjson::dom::object::iterator Class Reference

Public Member Functions

- const key_value_pair operator∗ () const noexcept

  Get the actual key/value pair.
- iterator & operator++ () noexcept

  Get the next key/value pair.
- bool operator!= (const iterator &other) const noexcept

  Check if these key value pairs come from the same place in the JSON.
- std::string_view key () const noexcept

  Get the key of this key/value pair.
- uint32_t key_length () const noexcept

  Get the length (in bytes) of the key in this key/value pair.
- bool key_equals (const std::string_view &o) const noexcept

  Returns true if the key in this key/value pair is equal to the provided string_view.
- bool key_equals_case_insensitive (const std::string_view &o) const noexcept

  Returns true if the key in this key/value pair is equal to the provided string_view in a case-insensitive manner.
- const char ∗ key_c_str () const noexcept

  Get the key of this key/value pair.
- element value () const noexcept

  Get the value of this key/value pair.

11.7.1 Detailed Description

Definition at line 25 of file object.h.

11.7.2 Member Function Documentation

11.7.2.1 key_equals()

```
bool simdjson::dom::object::iterator::key_equals (
            const std::string_view & o ) const  [inline], [noexcept]
```

Returns true if the key in this key/value pair is equal to the provided string_view.

Design notes: Instead of constructing a string_view and then comparing it with a user-provided strings, it is probably more performant to have dedicated functions taking as a parameter the string we want to compare against and return true when they are equal.

That avoids the creation of a temporary std::string_view. Though it is possible for the compiler to avoid entirely any overhead due to string_view, relying too much on compiler magic is problematic: compiler magic sometimes fail, and then what do you do? Also, enticing users to rely on high-performance function is probably better on the long run.

Definition at line 183 of file object.h.

11.7.2.2 key_equals_case_insensitive()

```
bool simdjson::dom::object::iterator::key_equals_case_insensitive (
            const std::string_view & o ) const  [inline], [noexcept]
```

Returns true if the key in this key/value pair is equal to the provided string_view in a case-insensitive manner.

Case comparisons may only be handled correctly for ASCII strings.

Definition at line 194 of file object.h.

11.7.2.3 key_length()

```
uint32_t simdjson::dom::object::iterator::key_length ( ) const  [inline], [noexcept]
```

Get the length (in bytes) of the key in this key/value pair.

You should expect this function to be faster than key().size().

Definition at line 160 of file object.h.

11.7.2.4 operator"!=()

```
bool simdjson::dom::object::iterator::operator!= (
            const iterator & other ) const  [inline], [noexcept]
```

Check if these key value pairs come from the same place in the JSON.

Part of the std::iterator interface.

Definition at line 149 of file object.h.

11.7.2.5 operator++()

```
object::iterator & simdjson::dom::object::iterator::operator++ ( )  [inline], [noexcept]
```

Get the next key/value pair.

Part of the std::iterator interface.

Definition at line 152 of file object.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/object.h

11.8 simdjson::dom::document_stream::iterator Class Reference

An iterator through a forward-only stream of documents.

Public Member Functions

- really_inline simdjson_result< element > operator* () noexcept

  Get the current document (or error).
- iterator & operator++ () noexcept

  Advance to the next document.
- really_inline bool operator!= (const iterator &other) const noexcept

  Check if we're at the end yet.

11.8.1 Detailed Description

An iterator through a forward-only stream of documents.

Definition at line 94 of file document_stream.h.

11.8.2 Member Function Documentation

11.8.2.1 operator"!=()

```
really_inline bool simdjson::dom::document_stream::iterator::operator!= (
             const iterator & other ) const  [noexcept]
```

Check if we're at the end yet.

Parameters

other	the end iterator to compare to.

Definition at line 124 of file document_stream.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/document_stream.h

11.9 simdjson::dom::key_value_pair Class Reference

Key/value pair in an object.

Public Attributes

- std::string_view key

  key in the key-value pair
- element value

  value in the key-value pair

11.9.1 Detailed Description

Key/value pair in an object.

Definition at line 191 of file object.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/object.h

11.10 simdjson::minifier< T > Class Template Reference

Minifies a JSON element or document, printing the smallest possible valid JSON.

Public Member Functions

- minifier (const T &_value) noexcept

  Create a new minifier.
- operator std::string () const noexcept

  Minify JSON to a string.
- std::ostream & print (std::ostream &out)

  Minify JSON to an output stream.
- std::ostream & **print** (std::ostream &out)
- std::ostream & **print** (std::ostream &out)
- std::ostream & **print** (std::ostream &out)
- really_inline std::ostream & **print** (std::ostream &out)
- std::ostream & **print** (std::ostream &out)
- std::ostream & **print** (std::ostream &out)
- std::ostream & **print** (std::ostream &out)

11.10.1 Detailed Description

template<typename T>
class simdjson::minifier< T >

Minifies a JSON element or document, printing the smallest possible valid JSON.

dom::parser parser; element doc = parser.parse(" [1 , 2 , 3] "_padded); cout << minify(doc) << endl; // prints [1,2,3]

Definition at line 38 of file minify.h.

11.10.2 Constructor & Destructor Documentation

11.10.2.1 minifier()

```
template<typename T >
simdjson::minifier< T >::minifier (
            const T & _value ) [inline], [noexcept]
```

Create a new minifier.

Parameters

_value	The document or element to minify.

Definition at line 45 of file minify.h.

The documentation for this class was generated from the following file:

- include/simdjson/minify.h

11.11 simdjson::dom::object Class Reference

JSON object.

Classes

- class iterator

Public Member Functions

- really_inline object () noexcept

  Create a new, invalid object.
- iterator begin () const noexcept

  Return the first key/value pair.
- iterator end () const noexcept

  One past the last key/value pair.
- size_t size () const noexcept

  Get the size of the object (number of keys).
- simdjson_result< element > operator[ ] (const std::string_view &key) const noexcept

  Get the value associated with the given key.
- simdjson_result< element > operator[ ] (const char ∗key) const noexcept

  Get the value associated with the given key.
- simdjson_result< element > at (const std::string_view &json_pointer) const noexcept

  Get the value associated with the given JSON pointer.
- simdjson_result< element > at_key (const std::string_view &key) const noexcept

  Get the value associated with the given key.
- simdjson_result< element > at_key_case_insensitive (const std::string_view &key) const noexcept

  Get the value associated with the given key in a case-insensitive manner.

11.11.1 Detailed Description

JSON object.

Definition at line 20 of file object.h.

11.11.2 Member Function Documentation

11.11.2.1 at()

```
simdjson_result< element > simdjson::dom::object::at (
            const std::string_view & json_pointer ) const  [inline], [noexcept]
```

Get the value associated with the given JSON pointer.

dom::parser parser; object obj = parser.parse(R"({ "foo": { "a": [10, 20, 30] }})"_padded); obj.at("foo/a/1") == 20 obj.at("foo")["a"].at(1) == 20

Returns

The value associated with the given JSON pointer, or:

- NO_SUCH_FIELD if a field does not exist in an object
- INDEX_OUT_OF_BOUNDS if an array index is larger than an array length
- INCORRECT_TYPE if a non-integer is used to access an array
- INVALID_JSON_POINTER if the JSON pointer is invalid and cannot be parsed

Definition at line 83 of file object.h.

11.11.2.2 at_key()

```
simdjson_result< element > simdjson::dom::object::at_key (
            const std::string_view & key ) const  [inline], [noexcept]
```

Get the value associated with the given key.

The key will be matched against **unescaped** JSON:

dom::parser parser; parser.parse(R"({ "a
": 1 })"_padded)["a\n"].get<uint64_t>().first == 1 parser.parse(R"({ "a
": 1 })"_padded)["a\\n"].get<uint64_t>().error() == NO_SUCH_FIELD

This function has linear-time complexity: the keys are checked one by one.

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object

Definition at line 120 of file object.h.

11.11.2.3 at_key_case_insensitive()

```
simdjson_result< element > simdjson::dom::object::at_key_case_insensitive (
            const std::string_view & key ) const [inline], [noexcept]
```

Get the value associated with the given key in a case-insensitive manner.

It is only guaranteed to work over ASCII inputs.

Note: The key will be matched against **unescaped** JSON.

This function has linear-time complexity: the keys are checked one by one.

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object

Definition at line 132 of file object.h.

11.11.2.4 begin()

```
object::iterator simdjson::dom::object::begin ( ) const [inline], [noexcept]
```

Return the first key/value pair.

Part of the std::iterable interface.

Definition at line 67 of file object.h.

11.11.2.5 end()

```
object::iterator simdjson::dom::object::end ( ) const [inline], [noexcept]
```

One past the last key/value pair.

Part of the std::iterable interface.

Definition at line 70 of file object.h.

11.11.2.6 operator[]() [1/2]

```
simdjson_result< element > simdjson::dom::object::operator[] (
             const char * key ) const  [inline], [noexcept]
```

Get the value associated with the given key.

The key will be matched against **unescaped** JSON:

dom::parser parser; parser.parse(R"({ "a
": 1 })"_padded)["a\n"].get<uint64_t>().first == 1 parser.parse(R"({ "a
": 1 })"_padded)["a\\n"].get<uint64_t>().error() == NO_SUCH_FIELD

This function has linear-time complexity: the keys are checked one by one.

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object
- INCORRECT_TYPE if this is not an object

Definition at line 80 of file object.h.

11.11.2.7 operator[]() [2/2]

```
simdjson_result< element > simdjson::dom::object::operator[] (
             const std::string_view & key ) const  [inline], [noexcept]
```

Get the value associated with the given key.

The key will be matched against **unescaped** JSON:

dom::parser parser; parser.parse(R"({ "a
": 1 })"_padded)["a\n"].get<uint64_t>().first == 1 parser.parse(R"({ "a
": 1 })"_padded)["a\\n"].get<uint64_t>().error() == NO_SUCH_FIELD

This function has linear-time complexity: the keys are checked one by one.

Returns

The value associated with this field, or:

- NO_SUCH_FIELD if the field does not exist in the object
- INCORRECT_TYPE if this is not an object

Definition at line 77 of file object.h.

11.11.2.8 size()

```
size_t simdjson::dom::object::size ( ) const  [inline], [noexcept]
```

Get the size of the object (number of keys).

It is a saturated value with a maximum of 0xFFFFFF: if the value is 0xFFFFFF then the size is 0xFFFFFF or greater.

Definition at line 73 of file object.h.

The documentation for this class was generated from the following file:

- include/simdjson/dom/object.h

11.12 simdjson::padded_string Struct Reference

String with extra allocation for ease of use with parser::parse()

Public Member Functions

- padded_string () noexcept
  Create a new, empty padded string.
- padded_string (size_t length) noexcept
  Create a new padded string buffer.
- padded_string (const char ∗data, size_t length) noexcept
  Create a new padded string by copying the given input.
- padded_string (const std::string &str_) noexcept
  Create a new padded string by copying the given input.
- padded_string (std::string_view sv_) noexcept
  Create a new padded string by copying the given input.
- padded_string (padded_string &&o) noexcept
  Move one padded string into another.
- padded_string & operator= (padded_string &&o) noexcept
  Move one padded string into another.
- void **swap** (padded_string &o) noexcept
- size_t size () const noexcept
  The length of the string.
- size_t length () const noexcept
  The length of the string.
- const char ∗ data () const noexcept
  The string data.
- char ∗ data () noexcept
  The string data.
- operator std::string_view () const
  Create a std::string_view with the same content.

Static Public Member Functions

- static simdjson_result< padded_string > load (const std::string &path) noexcept

  Load this padded string from a file.

11.12.1 Detailed Description

String with extra allocation for ease of use with parser::parse()

This is a move-only class, it cannot be copied.

Definition at line 20 of file padded_string.h.

11.12.2 Constructor & Destructor Documentation

11.12.2.1 padded_string() [1/5]

```
simdjson::padded_string::padded_string (
             size_t length )  [inline], [explicit], [noexcept]
```

Create a new padded string buffer.

Parameters

length	the size of the string.

Definition at line 38 of file padded_string.h.

11.12.2.2 padded_string() [2/5]

```
simdjson::padded_string::padded_string (
             const char * data,
             size_t length )  [inline], [explicit], [noexcept]
```

Create a new padded string by copying the given input.

Parameters

data	the buffer to copy
length	the number of bytes to copy

Definition at line 43 of file padded_string.h.

11.12.2.3 padded_string() [3/5]

```
simdjson::padded_string::padded_string (
            const std::string & str_ ) [inline], [noexcept]
```

Create a new padded string by copying the given input.

Parameters

*str_*	the string to copy

Definition at line 51 of file padded_string.h.

11.12.2.4 padded_string() [4/5]

```
simdjson::padded_string::padded_string (
            std::string_view sv_ ) [inline], [noexcept]
```

Create a new padded string by copying the given input.

Parameters

*sv_*	the string to copy

Definition at line 59 of file padded_string.h.

11.12.2.5 padded_string() [5/5]

```
simdjson::padded_string::padded_string (
            padded_string && o ) [inline], [noexcept]
```

Move one padded string into another.

The original padded string will be reduced to zero capacity.

Parameters

o	the string to move.

Definition at line 66 of file padded_string.h.

11.12.3 Member Function Documentation

11.12.3.1 length()

```
size_t simdjson::padded_string::length ( ) const  [inline], [noexcept]
```

The length of the string.

Does not include padding.

Definition at line 95 of file padded_string.h.

11.12.3.2 load()

```
simdjson_result< padded_string > simdjson::padded_string::load (
            const std::string & path )  [inline], [static], [noexcept]
```

Load this padded string from a file.

Parameters

path	the path to the file.

Definition at line 103 of file padded_string.h.

11.12.3.3 operator=()

```
padded_string & simdjson::padded_string::operator= (
            padded_string && o )  [inline], [noexcept]
```

Move one padded string into another.

The original padded string will be reduced to zero capacity.

Parameters

o	the string to move.

Definition at line 71 of file padded_string.h.

11.12.3.4 size()

```
size_t simdjson::padded_string::size ( ) const  [inline], [noexcept]
```

The length of the string.

Does not include padding.

Definition at line 93 of file padded_string.h.

The documentation for this struct was generated from the following file:

- include/simdjson/padded_string.h

11.13 simdjson::dom::parser Class Reference

A persistent document parser.

Public Member Functions

- really_inline parser (size_t max_capacity=SIMDJSON_MAXSIZE_BYTES) noexcept

  Create a JSON parser.
- really_inline parser (parser &&other) noexcept

  Take another parser's buffers and state.
- really_inline parser & operator= (parser &&other) noexcept

  Take another parser's buffers and state.
- ~parser ()=default

  Deallocate the JSON parser.
- simdjson_result< element > load (const std::string &path) &noexcept

  Load a JSON document from a file and return a reference to it.
- simdjson_result< element > **load** (const std::string &path) &&=delete
- simdjson_result< element > parse (const uint8_t ∗buf, size_t len, bool realloc_if_needed=true) &noexcept

  Parse a JSON document and return a temporary reference to it.
- simdjson_result< element > **parse** (const uint8_t ∗buf, size_t len, bool realloc_if_needed=true) &&=delete
- really_inline simdjson_result< element > **parse** (const char ∗buf, size_t len, bool realloc_if_needed=true) &noexcept
- really_inline simdjson_result< element > **parse** (const char ∗buf, size_t len, bool realloc_if_needed=true) &&=delete
- really_inline simdjson_result< element > **parse** (const std::string &s) &noexcept
- really_inline simdjson_result< element > **parse** (const std::string &s) &&=delete
- really_inline simdjson_result< element > **parse** (const padded_string &s) &noexcept
- really_inline simdjson_result< element > **parse** (const padded_string &s) &&=delete
- simdjson_result< document_stream > load_many (const std::string &path, size_t batch_size=DEFAULT_↩ BATCH_SIZE) noexcept

  Load a file containing many JSON documents.
- simdjson_result< document_stream > parse_many (const uint8_t ∗buf, size_t len, size_t batch_size=DE↩ FAULT_BATCH_SIZE) noexcept

  Parse a buffer containing many JSON documents.
- simdjson_result< document_stream > **parse_many** (const char ∗buf, size_t len, size_t batch_size=DEF↩ AULT_BATCH_SIZE) noexcept

- simdjson_result< document_stream > **parse_many** (const std::string &s, size_t batch_size=DEFAULT_↩ BATCH_SIZE) noexcept
- simdjson_result< document_stream > **parse_many** (const padded_string &s, size_t batch_size=DEFAU↩ LT_BATCH_SIZE) noexcept
- WARN_UNUSED error_code allocate (size_t capacity, size_t max_depth=DEFAULT_MAX_DEPTH) noexcept

  Ensure this parser has enough memory to process JSON documents up to `capacity` bytes in length and `max_↩ depth` depth.
- really_inline size_t capacity () const noexcept

  The largest document this parser can support without reallocating.
- really_inline size_t max_capacity () const noexcept

  The largest document this parser can automatically support.
- really_inline size_t max_depth () const noexcept

  The maximum level of nested object and arrays supported by this parser.
- really_inline void set_max_capacity (size_t max_capacity) noexcept

  Set max_capacity.

11.13.1 Detailed Description

A persistent document parser.

The parser is designed to be reused, holding the internal buffers necessary to do parsing, as well as memory for a single document. The parsed document is overwritten on each parse.

This class cannot be copied, only moved, to avoid unintended allocations.

Note

This is not thread safe: one parser cannot produce two documents at the same time!

Definition at line 36 of file parser.h.

11.13.2 Constructor & Destructor Documentation

11.13.2.1 parser() [1/2]

```
really_inline simdjson::dom::parser::parser (
            size_t max_capacity = SIMDJSON_MAXSIZE_BYTES )  [explicit], [noexcept]
```

Create a JSON parser.

The new parser will have zero capacity.

Parameters

max_capacity	The maximum document length the parser can automatically handle. The parser will allocate more capacity on an as needed basis (when it sees documents too big to handle) up to this amount. The parser still starts with zero capacity no matter what this number is: to allocate an initial capacity, call allocate() after constructing the parser. Defaults to SIMDJSON_MAXSIZE_BYTES (the largest single document simdjson can process).

Definition at line 18 of file parser.h.

11.13.2.2 parser() [2/2]

```
really_inline simdjson::dom::parser::parser (
            parser && other )  [default], [noexcept]
```

Take another parser's buffers and state.

Parameters

other	The parser to take. Its capacity is zeroed.

11.13.3 Member Function Documentation

11.13.3.1 allocate()

```
WARN_UNUSED error_code simdjson::dom::parser::allocate (
            size_t capacity,
            size_t max_depth = DEFAULT_MAX_DEPTH )  [inline], [noexcept]
```

Ensure this parser has enough memory to process JSON documents up to `capacity` bytes in length and `max←_depth` depth.

Parameters

capacity	The new capacity.
max_depth	The new max_depth. Defaults to DEFAULT_MAX_DEPTH.

Returns

The error, if there is one.

Definition at line 147 of file parser.h.

11.13.3.2 capacity()

```
really_inline size_t simdjson::dom::parser::capacity ( ) const  [noexcept]
```

The largest document this parser can support without reallocating.

Returns

Current capacity, in bytes.

Definition at line 136 of file parser.h.

11.13.3.3 load()

```
simdjson_result< element > simdjson::dom::parser::load (
            const std::string & path ) &  [inline], [noexcept]
```

Load a JSON document from a file and return a reference to it.

dom::parser parser; const element doc = parser.load("jsonexamples/twitter.json");

11.13.3.3.1 IMPORTANT: Document Lifetime The JSON document still lives in the parser: this is the most efficient way to parse JSON documents because it reuses the same buffers, but you *must* use the document before you destroy the parser or call parse() again.

11.13.3.3.2 Parser Capacity If the parser's current capacity is less than the file length, it will allocate enough capacity to handle it (up to max_capacity).

Parameters

path	The path to load.

Returns

The document, or an error:

- IO_ERROR if there was an error opening or reading the file.
- MEMALLOC if the parser does not have enough capacity and memory allocation fails.
- CAPACITY if the parser does not have enough capacity and len > max_capacity.
- other json errors if parsing fails.

Definition at line 79 of file parser.h.

11.13.3.4 load_many()

```
simdjson_result< document_stream > simdjson::dom::parser::load_many (
            const std::string & path,
            size_t batch_size = DEFAULT_BATCH_SIZE )  [inline], [noexcept]
```

Load a file containing many JSON documents.

dom::parser parser; for (const element doc : parser.load_many(path)) { cout << std::string(doc["title"]) << endl; }

11.13.3.4.1 Format

The file must contain a series of one or more JSON documents, concatenated into a single buffer, separated by whitespace. It effectively parses until it has a fully valid document, then starts parsing the next document at that point. (It does this with more parallelism and lookahead than you might think, though.)

documents that consist of an object or array may omit the whitespace between them, concatenating with no separator. documents that consist of a single primitive (i.e. documents that are not arrays or objects) MUST be separated with whitespace.

The documents must not exceed batch_size bytes (by default 1MB) or they will fail to parse. Setting batch_size to excessively large or excesively small values may impact negatively the performance.

11.13.3.4.2 Error Handling

All errors are returned during iteration: if there is a global error such as memory allocation, it will be yielded as the first result. Iteration always stops after the first error.

As with all other simdjson methods, non-exception error handling is readily available through the same interface, requiring you to check the error before using the document:

dom::parser parser; dom::document_stream docs; auto error = parser.load_many(path).get(docs); if (error) { cerr << error << endl; exit(1); } for (auto doc : docs) { std::string_view title; if ((error = doc["title"].get(title)) { cerr << error << endl; exit(1); } cout << title << endl; }

11.13.3.4.3 Threads

When compiled with SIMDJSON_THREADS_ENABLED, this method will use a single thread under the hood to do some lookahead.

11.13.3.4.4 Parser Capacity

If the parser's current capacity is less than batch_size, it will allocate enough capacity to handle it (up to max_capacity).

Parameters

path	File name pointing at the concatenated JSON to parse.
batch_size	The batch size to use. MUST be larger than the largest document. The sweet spot is cache-related: small enough to fit in cache, yet big enough to parse as many documents as possible in one tight loop. Defaults to 10MB, which has been a reasonable sweet spot in our tests.

Returns

The stream, or an error. An empty input will yield 0 documents rather than an EMPTY error. Errors:

- IO_ERROR if there was an error opening or reading the file.
- MEMALLOC if the parser does not have enough capacity and memory allocation fails.

- CAPACITY if the parser does not have enough capacity and batch_size > max_capacity.
- other json errors if parsing fails.

Definition at line 86 of file parser.h.

11.13.3.5 max_capacity()

```
really_inline size_t simdjson::dom::parser::max_capacity ( ) const  [noexcept]
```

The largest document this parser can automatically support.

The parser may reallocate internal buffers as needed up to this amount.

Returns

Maximum capacity, in bytes.

Definition at line 139 of file parser.h.

11.13.3.6 max_depth()

```
really_inline size_t simdjson::dom::parser::max_depth ( ) const  [noexcept]
```

The maximum level of nested object and arrays supported by this parser.

Returns

Maximum depth, in bytes.

Definition at line 142 of file parser.h.

11.13.3.7 operator=()

```
really_inline parser & simdjson::dom::parser::operator= (
             parser && other )  [default], [noexcept]
```

Take another parser's buffers and state.

Parameters

other	The parser to take. Its capacity is zeroed.

11.13.3.8 parse()

```
simdjson_result< element > simdjson::dom::parser::parse (
            const uint8_t * buf,
            size_t len,
            bool realloc_if_needed = true ) &  [inline], [noexcept]
```

Parse a JSON document and return a temporary reference to it.

dom::parser parser; element doc = parser.parse(buf, len);

11.13.3.8.1 IMPORTANT: Document Lifetime The JSON document still lives in the parser: this is the most efficient way to parse JSON documents because it reuses the same buffers, but you *must* use the document before you destroy the parser or call parse() again.

11.13.3.8.2 REQUIRED: Buffer Padding The buffer must have at least SIMDJSON_PADDING extra allocated bytes. It does not matter what those bytes are initialized to, as long as they are allocated.

If realloc_if_needed is true, it is assumed that the buffer does *not* have enough padding, and it is copied into an enlarged temporary buffer before parsing.

11.13.3.8.3 Parser Capacity If the parser's current capacity is less than len, it will allocate enough capacity to handle it (up to max_capacity).

Parameters

buf	The JSON to parse. Must have at least len + SIMDJSON_PADDING allocated bytes, unless realloc_if_needed is true.
len	The length of the JSON.
realloc_if_needed	Whether to reallocate and enlarge the JSON buffer to add padding.

Returns

The document, or an error:

- MEMALLOC if realloc_if_needed is true or the parser does not have enough capacity, and memory allocation fails.
- CAPACITY if the parser does not have enough capacity and len $>$ max_capacity.
- other json errors if parsing fails.

Definition at line 93 of file parser.h.

11.13.3.9 parse_many()

```
simdjson_result< document_stream > simdjson::dom::parser::parse_many (
            const uint8_t * buf,
            size_t len,
            size_t batch_size = DEFAULT_BATCH_SIZE )  [inline], [noexcept]
```

Parse a buffer containing many JSON documents.

dom::parser parser; for (element doc : parser.parse_many(buf, len)) { cout << std::string(doc["title"]) << endl; }

11.13.3.9.1 Format The buffer must contain a series of one or more JSON documents, concatenated into a single buffer, separated by whitespace. It effectively parses until it has a fully valid document, then starts parsing the next document at that point. (It does this with more parallelism and lookahead than you might think, though.)

documents that consist of an object or array may omit the whitespace between them, concatenating with no separator. documents that consist of a single primitive (i.e. documents that are not arrays or objects) MUST be separated with whitespace.

The documents must not exceed batch_size bytes (by default 1MB) or they will fail to parse. Setting batch_size to excessively large or excesively small values may impact negatively the performance.

11.13.3.9.2 Error Handling All errors are returned during iteration: if there is a global error such as memory allocation, it will be yielded as the first result. Iteration always stops after the first error.

As with all other simdjson methods, non-exception error handling is readily available through the same interface, requiring you to check the error before using the document:

dom::parser parser; dom::document_stream docs; auto error = parser.load_many(path).get(docs); if (error) { cerr << error << endl; exit(1); } for (auto doc : docs) { std::string_view title; if ((error = doc["title"].get(title)) { cerr << error << endl; exit(1); } cout << title << endl; }

11.13.3.9.3 REQUIRED: Buffer Padding The buffer must have at least SIMDJSON_PADDING extra allocated bytes. It does not matter what those bytes are initialized to, as long as they are allocated.

11.13.3.9.4 Threads When compiled with SIMDJSON_THREADS_ENABLED, this method will use a single thread under the hood to do some lookahead.

11.13.3.9.5 Parser Capacity If the parser's current capacity is less than batch_size, it will allocate enough capacity to handle it (up to max_capacity).

Parameters

buf	The concatenated JSON to parse. Must have at least len + SIMDJSON_PADDING allocated bytes.
len	The length of the concatenated JSON.
batch_size	The batch size to use. MUST be larger than the largest document. The sweet spot is cache-related: small enough to fit in cache, yet big enough to parse as many documents as possible in one tight loop. Defaults to 10MB, which has been a reasonable sweet spot in our tests.

Returns

The stream, or an error. An empty input will yield 0 documents rather than an EMPTY error. Errors:

- MEMALLOC if the parser does not have enough capacity and memory allocation fails
- CAPACITY if the parser does not have enough capacity and batch_size $>$ max_capacity.
- other json errors if parsing fails.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Definition at line 123 of file parser.h.

11.13.3.10 set_max_capacity()

```
really_inline void simdjson::dom::parser::set_max_capacity (
            size_t max_capacity )  [noexcept]
```

Set max_capacity.

This is the largest document this parser can automatically support.

The parser may reallocate internal buffers as needed up to this amount as documents are passed to it.

This call will not allocate or deallocate, even if capacity is currently above max_capacity.

Parameters

max_capacity	The new maximum capacity, in bytes.

Definition at line 191 of file parser.h.

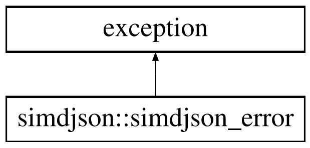
The documentation for this class was generated from the following file:

- include/simdjson/dom/parser.h

11.14 simdjson::simdjson_error Struct Reference

Exception thrown when an exception-supporting simdjson method is called.

Inheritance diagram for simdjson::simdjson_error:



Public Member Functions

- simdjson_error (error_code error) noexcept

  Create an exception from a simdjson error code.
- const char ∗ what () const noexcept

  The error message.
- error_code error () const noexcept

  The error code.

11.14.1 Detailed Description

Exception thrown when an exception-supporting simdjson method is called.

Definition at line 61 of file error.h.

11.14.2 Constructor & Destructor Documentation

11.14.2.1 simdjson_error()

```
simdjson::simdjson_error::simdjson_error (
            error_code error )  [inline], [noexcept]
```

Create an exception from a simdjson error code.

Parameters

error	The error code

Definition at line 66 of file error.h.

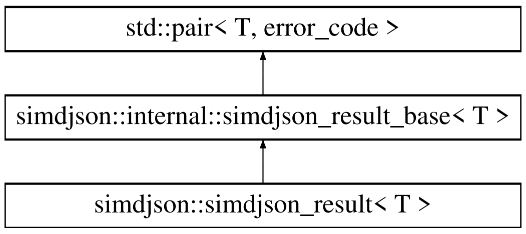
The documentation for this struct was generated from the following file:

- include/simdjson/error.h

11.15 simdjson::simdjson_result< T > Struct Template Reference

The result of a simdjson operation that could fail.

Inheritance diagram for simdjson::simdjson_result< T >:


std::pair< T, error_code >
simdjson::internal::simdjson_result_base< T >
simdjson::simdjson_result< T >

Public Member Functions

- really_inline void tie (T &value, error_code &error) &&noexcept

  Move the value and the error to the provided variables.
- WARN_UNUSED really_inline error_code get (T &value) &&noexcept

  Move the value to the provided variable.
- really_inline error_code error () const noexcept

  The error.
- really_inline T & value () noexcept(false)

  Get the result value.
- really_inline T && take_value () &&noexcept(false)

  Take the result value (move it).
- really_inline operator T&& () &&noexcept(false)

  Cast to the value (will throw on error).

11.15.1 Detailed Description

template<typename T>
struct simdjson::simdjson_result< T >

The result of a simdjson operation that could fail.

Gives the option of reading error codes, or throwing an exception by casting to the desired result.

Definition at line 175 of file error.h.

11.15.2 Member Function Documentation

11.15.2.1 get()

```
template<typename T >
WARN_UNUSED really_inline error_code simdjson::simdjson_result< T >::get (
            T & value ) &&  [noexcept]
```

Move the value to the provided variable.

Parameters

value	The variable to assign the value to. May not be set if there is an error.

Definition at line 112 of file error.h.

11.15.2.2 operator T&&()

```
template<typename T >
really_inline simdjson::simdjson_result< T >::operator T&&  [noexcept]
```

Cast to the value (will throw on error).

Exceptions

simdjson_error	if there was an error.

Definition at line 134 of file error.h.

11.15.2.3 take_value()

```
template<typename T >
really_inline T && simdjson::simdjson_result< T >::take_value  [noexcept]
```

Take the result value (move it).

Exceptions

simdjson_error	if there was an error.

Definition at line 129 of file error.h.

11.15.2.4 tie()

```
template<typename T >
really_inline void simdjson::simdjson_result< T >::tie (
             T & value,
             error_code & error ) &&  [noexcept]
```

Move the value and the error to the provided variables.

simdjson_result<T> inline implementation

Parameters

value	The variable to assign the value to. May not be set if there is an error.
error	The variable to assign the error to. Set to SUCCESS if there is no error.

Definition at line 107 of file error.h.

11.15.2.5 value()

```
template<typename T >
really_inline T & simdjson::simdjson_result< T >::value  [noexcept]
```

Get the result value.

Exceptions

simdjson_error	if there was an error.

Definition at line 124 of file error.h.

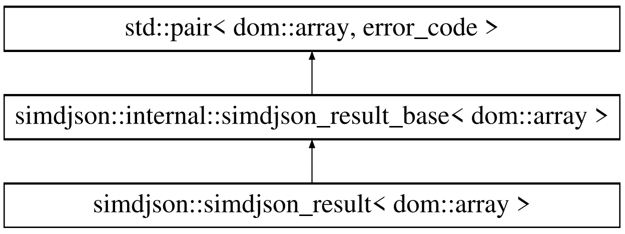
The documentation for this struct was generated from the following file:

- include/simdjson/error.h

11.16 simdjson::simdjson_result< dom::array > Struct Reference

The result of a JSON conversion that may fail.

Inheritance diagram for simdjson::simdjson_result< dom::array >:


std::pair< dom::array, error_code >
simdjson::internal::simdjson_result_base< dom::array >
simdjson::simdjson_result< dom::array >

Public Member Functions

- simdjson_result< dom::element > **at** (const std::string_view &json_pointer) const noexcept
- simdjson_result< dom::element > **at** (size_t index) const noexcept
- dom::array::iterator **begin** () const noexcept(false)
- dom::array::iterator **end** () const noexcept(false)
- size_t **size** () const noexcept(false)

11.16.1 Detailed Description

The result of a JSON conversion that may fail.

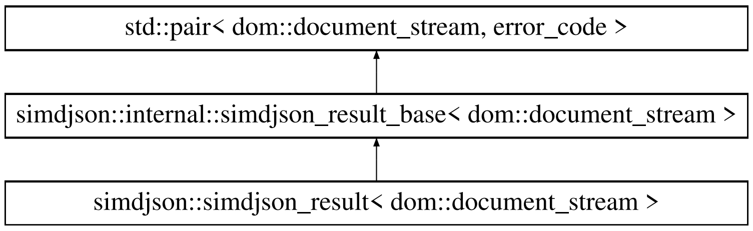
Definition at line 125 of file array.h.

The documentation for this struct was generated from the following file:

- include/simdjson/dom/array.h

11.17 simdjson::simdjson_result< dom::document_stream > Struct Reference

Inheritance diagram for simdjson::simdjson_result< dom::document_stream >:


std::pair< dom::document_stream, error_code >
simdjson::internal::simdjson_result_base< dom::document_stream >
simdjson::simdjson_result< dom::document_stream >

Public Member Functions

- really_inline dom::document_stream::iterator **begin** () noexcept(false)
- really_inline dom::document_stream::iterator **end** () noexcept(false)

11.17.1 Detailed Description

Definition at line 242 of file document_stream.h.

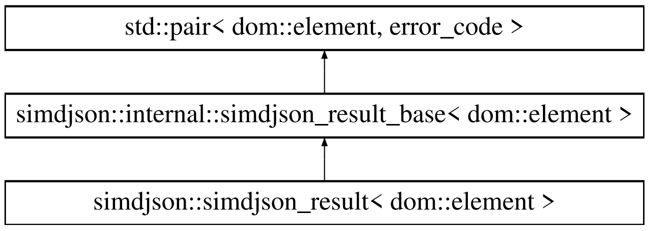
The documentation for this struct was generated from the following file:

- include/simdjson/dom/document_stream.h

11.18 simdjson::simdjson_result< dom::element > Struct Reference

The result of a JSON navigation that may fail.

Inheritance diagram for simdjson::simdjson_result< dom::element >:


std::pair< dom::element, error_code >
simdjson::internal::simdjson_result_base< dom::element >
simdjson::simdjson_result< dom::element >

Public Member Functions

- really_inline simdjson_result< dom::element_type > **type** () const noexcept
- template<typename T >
  really_inline bool **is** () const noexcept
- template<typename T >
  really_inline simdjson_result< T > **get** () const noexcept
- template<typename T >
  WARN_UNUSED really_inline error_code **get** (T &value) const noexcept
- really_inline simdjson_result< dom::array > **get_array** () const noexcept
- really_inline simdjson_result< dom::object > **get_object** () const noexcept
- really_inline simdjson_result< const char ∗ > **get_c_str** () const noexcept
- really_inline simdjson_result< size_t > **get_string_length** () const noexcept
- really_inline simdjson_result< std::string_view > **get_string** () const noexcept
- really_inline simdjson_result< int64_t > **get_int64** () const noexcept
- really_inline simdjson_result< uint64_t > **get_uint64** () const noexcept
- really_inline simdjson_result< double > **get_double** () const noexcept
- really_inline simdjson_result< bool > **get_bool** () const noexcept
- really_inline bool **is_array** () const noexcept
- really_inline bool **is_object** () const noexcept
- really_inline bool **is_string** () const noexcept
- really_inline bool **is_int64** () const noexcept
- really_inline bool **is_uint64** () const noexcept
- really_inline bool **is_double** () const noexcept
- really_inline bool **is_bool** () const noexcept
- really_inline bool **is_null** () const noexcept
- really_inline simdjson_result< dom::element > **operator[ ]** (const std::string_view &key) const noexcept
- really_inline simdjson_result< dom::element > **operator[ ]** (const char ∗key) const noexcept
- really_inline simdjson_result< dom::element > **at** (const std::string_view &json_pointer) const noexcept
- really_inline simdjson_result< dom::element > **at** (size_t index) const noexcept
- really_inline simdjson_result< dom::element > **at_key** (const std::string_view &key) const noexcept
- really_inline simdjson_result< dom::element > **at_key_case_insensitive** (const std::string_view &key) const noexcept
- really_inline **operator bool** () const noexcept(false)
- really_inline **operator const char ∗** () const noexcept(false)
- really_inline **operator std::string_view** () const noexcept(false)
- really_inline **operator uint64_t** () const noexcept(false)
- really_inline **operator int64_t** () const noexcept(false)
- really_inline **operator double** () const noexcept(false)
- really_inline **operator dom::array** () const noexcept(false)
- really_inline **operator dom::object** () const noexcept(false)
- really_inline dom::array::iterator **begin** () const noexcept(false)
- really_inline dom::array::iterator **end** () const noexcept(false)

11.18.1 Detailed Description

The result of a JSON navigation that may fail.

Definition at line 473 of file element.h.

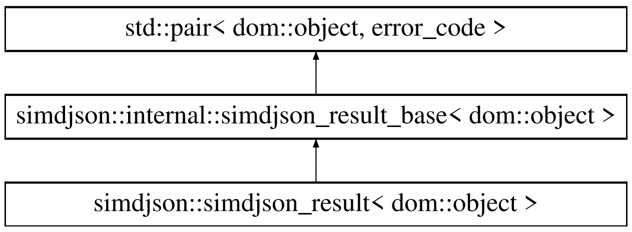
The documentation for this struct was generated from the following file:

- include/simdjson/dom/element.h

11.19 simdjson::simdjson_result< dom::object > Struct Reference

The result of a JSON conversion that may fail.

Inheritance diagram for simdjson::simdjson_result< dom::object >:


std::pair< dom::object, error_code >
simdjson::internal::simdjson_result_base< dom::object >
simdjson::simdjson_result< dom::object >

Public Member Functions

- simdjson_result< dom::element > **operator[ ]** (const std::string_view &key) const noexcept
- simdjson_result< dom::element > **operator[ ]** (const char ∗key) const noexcept
- simdjson_result< dom::element > **at** (const std::string_view &json_pointer) const noexcept
- simdjson_result< dom::element > **at_key** (const std::string_view &key) const noexcept
- simdjson_result< dom::element > **at_key_case_insensitive** (const std::string_view &key) const noexcept
- dom::object::iterator **begin** () const noexcept(false)
- dom::object::iterator **end** () const noexcept(false)
- size_t **size** () const noexcept(false)

11.19.1 Detailed Description

The result of a JSON conversion that may fail.

Definition at line 228 of file object.h.

The documentation for this struct was generated from the following file:

- include/simdjson/dom/object.h

Chapter 12

File Documentation

12.1 include/simdjson/simdjson.h File Reference

12.1.1 Detailed Description

Deprecated We'll be removing this file so it isn't confused with the top level simdjson.h