YAJL-Fort Documentation

Release 1.0

Neil N. Carlson

Mar 24, 2018

Contents:

The YAJL-Fort package provides a modern object-oriented Fortran interface to the YAJL C library, which is an event-driven parser for JSON data streams. JSON is an open standard data interchange format. It is light weight, flexible, easy for humans to read and write, and language independent.

Note that unlike most other JSON libraries, YAJL does not provide or impose an in-memory data structure representation of the JSON data. That is left to higher-level application code through custom parsing event callback functions.

Also included in YAJL-Fort is a module that defines data structures for representing arbitrary JSON data, and procedures built on the YAJL interface for populating the data structures with JSON data read from a file or string.

YAJL-Fort is open source software, distributed under the MIT license.

Get YAJL-Fort on GitHub: https://github.com/nncarlson/yajl-fort

CHAPTER 1

Building

Prerequisites:

- CMake version 3.1 or newer.
- YAJL 2.0.1 or newer. You need both the library and the header files. This is a standard binary package in most any Linux distribution. Note that the header files are often in a separate "devel" package. The source can be downloaded from https://github.com/lloyd/yajl/releases if you must build the library yourself.
- A Fortran compiler that supports the 2003/2008 features used by YAJL-Fort, and its companion (or a compatible) C compiler.

Clone the YAJL-Fort source repository with SSH:

```
git clone git@github.com:nncarlson/yajl-fort.git
```

or with HTTPS:

```
git clone https://github.com/nncarlson/yajl-fort.git
```

Set your `FC` and `CC` environment variables to your Fortran and C compilers. If YAJL is not in a standard location you will also need to set your `YAJL_ROOT` environment variable to the root directory of the YAJL installation. Then create a build directory and run `cmake` from the directory:

```
cd yajl-fort
mkdir build
cd build
cmake ..
```

CMake should find your compilers with the help of the `FC` and `CC` variables. The default is to configure a `Release` build. Some CMake variables you might want to set on the `cmake` command line:

- `CMAKE_BUILD_TYPE`: to specify a `Debug` build type, for example
- `CMAKE_Fortran_FLAGS`: additional Fortran compiler flags
- `CMAKE_INSTALL_PREFIX`: where to install the library and module files

Then compile the library and tests, and run the tests (all should pass):

```
make
ctest
```

Then see the documentation for the *yajl_fort* and *json* modules and the examples therein.

1.1 Compiler status and notes

The following compilers are known to work:

- NAG 5.3.2, 6.0, 6.1, and 6.2
- Intel 16.0.2, 17.0.6, 18.0.1
- GFortran 6.4.1, 7.2.1, 7.3.1, 8.0.1 (20180311 trunk)
- IBM xlf 15.1.6 (use the xlf2008 executable)

The following compilers are known to **not** work:

- Flang (5.0.1, b356fc9b, 20180316 master)
- Any PGI up to and including 18.1.1
  - `yajl_fort` module may be usable for 18.1.1, but no earlier version.
  - See test case pgi-20180320.f90 for bug affecting `json` module.

The `CMakeLists.txt` file has special stanzas for some compilers that set specific flags that are known to be needed. If you are using another compiler it too may need some special flags. These can be set on the `cmake` command line with `CMAKE_Fortran_FLAGS` or a stanza can be added to the `CMakeLists.txt` file.

CHAPTER 2

The yajl_fort module

The `yajl_fort` module defines an object-oriented Fortran interface to the YAJL C library, which is an event-driven parser for JSON data streams. JSON is an open standard data interchange format. It is lightweight, flexible, easy for humans to read and write, and language independent.

Note: Unlike most other JSON libraries, YAJL does not provide or impose an in-memory data representation, but instead uses callbacks to accommodate any in-memory representation. The same is true of `yajl_fort`, being only an interface to YAJL. If you want an in-memory representation (and you most likely do), you may do so using `yajl_fort`, but you provide the code that defines and populates the in-memory representation using the callbacks according to your specific requirements.

2.1 Synopsis

```
use yajl_fort
```

Derived types `fyajl_callbacks` (abstract), `fyajl_parser`, `fyajl_status`

Functions `fyajl_get_error`, `fyajl_status_to_string`

Parameters

- callback return: `FYAJL_CONTINUE_PARSING`, `FYAJL_TERMINATE_PARSING`
- kind: `FYAJL_INTEGER_KIND`, `FYAJL_REAL_KIND`
- parser return: `FYAJL_STATUS_OK`, `FYAJL_STATUS_ERROR`, `FYAJL_STATUS_CLIENT_CANCELED`
- option: `FYAJL_ALLOW_COMMENTS`, `FYAJL_ALLOW_MULTIPLE_DOCUMENTS`, `FYAJL_ALLOW_PARTIAL_DOCUMENT`, `FYAJL_ALLOW_TRAILING_GARBAGE`, `FYAJL_DONT_VALIDATE_STRINGS`

2.1.1 Prerequisites

The `yajl_fort` module uses YAJL version 2.0 or later. The source code for this library can be downloaded from https://github.com/lloyd/yajl/releases. The library is also available as a standard binary package in all major Linux distributions. See http://lloyd.github.io/yajl/ for additional information.

2.2 Parser callback functions

2.2.1 JSON overview

The JSON data language is quite simple. It is built on two basic data structures. An *array* is an ordered list of comma-separated *values* enclosed in brackets (`[` and `]`). An *object* is an unordered list of comma-separated *name* **:** *value* pairs enclosed in braces (`{` and `}`). A *name* is a string enclosed in double quotes, and a *value* is one of the following: a string in double quotes, a number (integer or real), a boolean literal (`true` or `false`), the literal `null`, or an *object* or *array*. Note how the data structures can be nested. Whitespace is insignificant except in strings. At the outermost level, what is considered valid JSON text varies between the several standard documents, and it comes down to a matter of agreement between the producer and consumer of the data. Originally it was required to be an *object* or *array*, but more recently any JSON *value* is considered valid. The YAJL library follows the latter. See this blog post for a discussion of the issue, and http://www.json.org for a detailed description of the JSON syntax.

2.2.2 The callbacks derived type

The C language YAJL parser operates by calling application-defined callback functions in response to the various events encountered while parsing the input stream. The callback functions communicate with each other through a common, application-defined, context data struct, and a void pointer to that data struct is passed to each of the callbacks. In this Fortran interface, this application-defined code/data is implemented by the abstract derived type `fyajl_callbacks`:

```
type, abstract :: fyajl_callbacks
contains
  procedure(cb_no_args), deferred :: start_map
  procedure(cb_no_args), deferred :: end_map
  procedure(cb_string),  deferred :: map_key
  procedure(cb_no_args), deferred :: null_value
  procedure(cb_logical), deferred :: logical_value
  procedure(cb_integer), deferred :: integer_value
  procedure(cb_double),  deferred :: double_value
  procedure(cb_string),  deferred :: string_value
  procedure(cb_no_args), deferred :: start_array
  procedure(cb_no_args), deferred :: end_array
end type fyajl_callbacks
```

Application code extends this type, adding the desired context data components and providing concrete implementations of the callback functions. The required interfaces for the deferred type bound callback functions are:

```
integer function cb_no_args(this)
  class(fyajl_callbacks) :: this
integer function cb_integer(this, value)
  class(fyajl_callbacks) :: this
  integer(FYAJL_INTEGER_KIND), intent(in) :: value
integer function cb_double(this, value)
  class(fyajl_callbacks) :: this
  real(FYAJL_REAL_KIND), intent(in) :: value
```

```
integer function cb_logical(this, value)
  class(fyajl_callbacks) :: this
  logical, intent(in) :: value
integer function cb_string(this, value)
  class(fyajl_callbacks) :: this
  character(*,kind=c_char), intent(in) :: value
```

The return value of each function must be either of the module parameters `FYAJL_CONTINUE_PARSING` or `FYAJL_TERMINATE_PARSING`. The latter return value will cause the parser to terminate with an error. The module kind parameters for integer and real values, `FYAJL_INTEGER_KIND` and `FYAJL_REAL_KIND`, correspond to C's `long long` and `double`, and are dictated by the YAJL library. The callbacks are invoked as follows:

start_map called when a { is parsed, marking the start of an *object*

end_map called when a } is parsed, marking the end of an *object*.

start_array called when a [is parsed, marking the start of an *array*.

end_array called when a] is parsed, marking the end of an *array*.

map_key called when the *name* of a *name* **:** *value* pair is parsed, and the parsed name string is passed to the function.

integer_value called when an integer *value* is parsed, and the value is passed to the function.

double_value called when a real *value* is parsed, and the value is passed to the function.

string_value called when a string *value* is parsed, and the value is passed to the function.

logical_value called when the *value* token `true` or `false` is parsed, and the corresponding Fortran logical value is passed to the function.

null_value called when the *value* token `null` is parsed.

2.3 Parsing

The derived type `fyajl_parser` and its type bound procedures implement the JSON parser. First, as described in the previous section, an application-specific extension of the abstract type `fyajl_callbacks` must be defined and an instance (here `callbacks`) of that extension initialized:

```
type, extends(fyajl_callbacks) :: my_callbacks
  ! context data defined here
contains
  ! define the deferred type bound procedures
end type
type(my_callbacks), target :: callbacks
! initialize the context data of callbacks as needed
```

The parser is then initialized by passing the `callbacks` object to its `init` subroutine:

```
type(fyajl_parser) :: parser
call parser%init(callbacks)
```

Note that proper finalization of the parser object occurs automatically when the object is deallocated or otherwise ceases to exist. Finalization of the callback object is the responsibility of the application.

Parsing is carried out incrementally via repeated calls to the `parse` method:

```
call parser%parse(buffer, stat)
  character(kind=c_char), intent(in) :: buffer(:)
  type(fyajl_status), intent(out) :: stat
```

Successive chunks of the JSON text are passed in the `buffer` array, and the parsing status is returned in `stat`; see *Error handling*.

After all the JSON text has been fed to the parser, the `parse_complete` method must be called to parse any internally buffered JSON text that might remain:

```
call parser%parse_complete(stat)
  type(fyajl_status), intent(out) :: stat
```

This is required because the parser is stream based and it needs an explicit end-of-input signal to force it to parse content at the end of the stream that sometimes exists. The parsing status is returned in `stat`; see *Error handling*.

The function call `parser%bytes_consumed()` returns the number of characters consumed from `buffer` in the last call to `parse`.

2.3.1 Error handling

The `parse` and `parse_complete` methods return a `type(fyajl_status)` status value, which equals one of the following module parameters:

FYAJL_STATUS_OK No error.

FYAJL_STATUS_ERROR A parsing error was encountered; use `fyajl_get_error` to get information about it.

FYAJL_STATUS_CLIENT_CANCELLED One of the callback procedures returned `FYAJL_TERMINATE_PARSING`.

The comparison operators == and /= are defined for `type(fyajl_status)` values.

Several additional functions (not type bound) are provided for error handling.

```
fyajl_get_error(parser, verbose, buffer)
  logical, intent(in) :: verbose
  character(kind=c_char), intent(in) :: buffer(:)
```

Returns a character string describing the the error encountered by the parser. If `verbose` is true, the message will include the portion of the input stream where the error occurred together with an arrow pointing to the specific character. The `buffer` array should contain the chunk of JSON text passed in the last call to `parse`.

```
fyajl_status_to_string(code)
  type(fyajl_status), intent(in) :: code
```

Returns a character string describing the specified status value `code`.

2.3.2 Parsing options

The parser supports several options provided by the YAJL library. They are set and unset using the `set_option` and `unset_option` methods after the parser has been initialized:

```
call parser%set_option(option)
call parser%unset_option(option)
```

where `option` is one of the following module parameters. The default for all is unset.

FYAJL_ALLOW_COMMENTS JSON does not allow for comments. Setting this option causes the parser to ignore javascript style comments in the input stream. This includes single-line comments that begin with // and continue to the end of the line. This is a very useful extention to the JSON standard, but one that is not supported by many JSON parsers.

FYAJL_DONT_VALIDATE_STRINGS By default, the parser verifies that all strings are valid UTF-8. This option disables this check, resulting in slightly faster parsing.

FYAJL_ALLOW_TRAILING_GARBAGE By default, `parse_complete` verifies that the entire input text has been consumed and will return an error if it finds otherwise. Setting this option will disable this check. This can be useful when parsing an input stream that contains more than one JSON document. In such scenarios, the `bytes_consumed` method is useful for identifying the trailing portion of the input text for subsequent handling.

FYAJL_ALLOW_MULTIPLE_DOCUMENTS An instance of a parser normally expects that the input stream consists of a single JSON document. Setting this option changes that behavior and allows an instance to parse an input stream containing multiple documents that are separated by whitespace.

FYAJL_ALLOW_PARTIAL_DOCUMENT By default, `parse_complete` verifies that the top level *object* is complete; that is, the closing } has been parsed. If it finds otherwise it returns an error. Setting this option disables this check.

2.4 Examples

In addition to the simple example presented below, here are some links to genuine uses of `yajl_fort`:

- The *json module* included in YAJL-Fort defines structures for in-memory representation of arbitrary JSON data, and procedures for populating the structures with JSON data read from a file or string using `yajl_fort`.
- The `parameter_list_type` module from the Petaca library defines a hierarchical data structure that is very similar to JSON, but that is much better suited to Fortran use. A subset of JSON maps naturally to this data structure, and the parameter_list_json module provides procedures built on `yajl_fort` for populating this structure with JSON data read from a file or string. This illustrates a major advantage of the customized callback approach, in that the callbacks implement the grammar of this JSON subset so that syntax errors are detected promptly during parsing.

2.4.1 A JSON white space stripper

This simple program reads JSON text from a file, strips all insignificant white space from it, including newlines, and writes the result to standard output. Somewhat contrived, but it serves to illustrate how to use `yajl_fort` in a complete program. No in-memory representation of the JSON data is needed in this case; it is streamed to the output as it is being parsed. The only slightly complicated aspect, requiring some context data, is keeping track of when the `,` separator needs to be written. The source for this example is in test/strip.f90

The module `strip_cb_type` defines the callback structure. The callback functions merely echo their respective token to the output. However the `*_value` and `map_key` functions must first write a `,` if the value follows a value in an array list, or if the key follows a key:value pair in an object list. The hierarchical structure of JSON means that at any moment of the parsing there may be multiple array or object lists in the process of being parsed. To keep track for each list of whether a comma is needed or not, we use a stack. Here we just use a fixed length logical array `comma` and an integer index `top` that points to the top of the stack. These are the common context data shared by the callbacks. The subroutines `push`, `pop`, and `write_comma` take care of managing the stack.

```
module strip_cb_type

  use,intrinsic :: iso_fortran_env, only: output_unit
  use yajl_fort
  implicit none
  private

  type, extends(fyajl_callbacks), public :: strip_cb
    integer :: top = 1
    logical :: comma(99) = .false.
  contains
    procedure :: start_map
    procedure :: end_map
    procedure :: map_key
    procedure :: null_value
    procedure :: logical_value
    procedure :: integer_value
    procedure :: double_value
    procedure :: string_value
    procedure :: start_array
    procedure :: end_array
  end type

contains

  subroutine push(this)
    class(strip_cb), intent(inout) :: this
    this%top = this%top + 1
    this%comma(this%top) = .false. ! start of new list
  end subroutine

  subroutine pop(this)
    class(strip_cb), intent(inout) :: this
    this%top = this%top - 1
  end subroutine

  subroutine write_comma(this, next)
    class(strip_cb), intent(inout) :: this
    logical, intent(in) :: next
    if (this%comma(this%top)) write(output_unit,'(",")',advance='no')
    this%comma(this%top) = next
  end subroutine

  integer function null_value(this) result(stat)
    class(strip_cb) :: this
    call write_comma(this, next=.true.)
    write(output_unit,'("null")',advance='no')
    stat = FYAJL_CONTINUE_PARSING
  end function

  integer function logical_value(this, value) result(stat)
    class(strip_cb) :: this
    logical, intent(in) :: value
    call write_comma(this, next=.true.)
    if (value) then
      write(output_unit,'("true")',advance='no')
    else
      write(output_unit,'("false")',advance='no')
    end if
    stat = FYAJL_CONTINUE_PARSING
  end function
```

```
integer function integer_value(this, value) result(stat)
  class(strip_cb) :: this
  integer(fyajl_integer_kind), intent(in) :: value
  call write_comma(this, next=.true.)
  write(output_unit,'(i0)',advance='no') value
  stat = FYAJL_CONTINUE_PARSING
end function

integer function double_value(this, value) result(stat)
  class(strip_cb) :: this
  real(fyajl_real_kind), intent(in) :: value
  call write_comma(this, next=.true.)
  write(output_unit,'(g0)',advance='no') value
  stat = FYAJL_CONTINUE_PARSING
end function

integer function string_value(this, value) result(stat)
  class(strip_cb) :: this
  character(*), intent(in) :: value
  call write_comma(this, next=.true.)
  write(output_unit,'(3a)',advance='no') '"', value, '"'
  stat = FYAJL_CONTINUE_PARSING
end function

integer function map_key(this, value) result(stat)
  class(strip_cb) :: this
  character(*), intent(in) :: value
  call write_comma(this, next=.false.) ! no comma for next value
  write(output_unit,'(3a)',advance='no') '"', value, '":'
  stat = FYAJL_CONTINUE_PARSING
end function

integer function start_map(this) result(stat)
  class(strip_cb) :: this
  call write_comma(this, next=.true.)
  write(output_unit,'("{")',advance='no')
  call push(this)  ! starting new list
  stat = FYAJL_CONTINUE_PARSING
end function

integer function end_map(this) result(stat)
  class(strip_cb) :: this
  write(output_unit,'("}")',advance='no')
  call pop(this) ! finished this list
  stat = FYAJL_CONTINUE_PARSING
end function

integer function start_array(this) result(stat)
  class(strip_cb) :: this
  call write_comma(this, next=.true.)
  write(output_unit,'("[")',advance='no')
  call push(this) ! starting new list
  stat = FYAJL_CONTINUE_PARSING
end function

integer function end_array(this) result(stat)
  class(strip_cb) :: this
```

```
    write(output_unit,'("]")',advance='no')
    call pop(this) ! finished this list
    stat = FYAJL_CONTINUE_PARSING
  end function

end module
```

The main program opens the file specified on the command line for unformatted stream input, and then reads and parses buffer-sized chunks until the whole file has been read. This is a pattern most any use of `yajl_fort` will follow.

```
program strip_json

  use,intrinsic :: iso_fortran_env
  use yajl_fort
  use strip_cb_type
  implicit none

  integer :: ios, lun, last_pos, curr_pos, buflen
  character(64) :: arg
  character(:), allocatable :: file
  character :: buffer(64) ! intentionally small buffer for testing
  type(strip_cb), target :: callbacks
  type(fyajl_parser), target :: parser
  type(fyajl_status) :: stat

  !! Get the file name from the command line
  if (command_argument_count() == 1) then
    call get_command_argument(1, arg)
    file = trim(arg)
  else
    call get_command(arg)
    write(error_unit,'(a)') 'usage: ' // trim(arg) // ' file'
    stop
  end if

  !! Open the file for stream input
  open(newunit=lun,file=file,action='read',access='stream')
  inquire(lun,pos=last_pos)

  !! Initialize the parser with our callback functions
  call parser%init(callbacks)
  call parser%set_option(FYAJL_ALLOW_COMMENTS)

  do
    !! Read the next chunk of the input file
    read(lun,iostat=ios) buffer
    if (ios /= 0 .and. ios /= iostat_end) then
      write(error_unit,'(a,i0)') 'read error: iostat=', ios
      exit
    end if

    !! Feed the chunk to the parser and check for errors.
    inquire(lun,pos=curr_pos)
    buflen = curr_pos - last_pos
    last_pos = curr_pos
    if (buflen > 0) then
      call parser%parse(buffer(:buflen), stat)
      if (stat /= FYAJL_STATUS_OK) then
        write(error_unit,'(a)') &
           fyajl_get_error(parser, .true., buffer(:buflen))
        exit
      end if
    end if

    !! If there are no more chunks to read, tell the parser.
    if (ios == iostat_end) then
      call parser%complete_parse(stat)
      if (stat /= FYAJL_STATUS_OK) then
        write(error_unit,'(a)') &
           fyajl_get_error(parser, .false., buffer(:buflen))
      end if
      exit
    end if
  end do
  close(lun)

end program
```

CHAPTER 3

The json module

The `json` module defines derived data types for representing arbitrary JSON data, and procedures for instantiating objects of those types from JSON text read from a file or string.

This module uses *yajl_fort* for parsing the JSON input data.

Note: This module is a work-in-progress. While it provides the ability to read arbitrary JSON data and represent it in memory, it lacks many convenient methods for working with the data. Needed in particular, are methods for direct access to values using a "path" type of indexing.

3.1 Usage

Refer to http://www.json.org for a detailed description of the JSON syntax. The derived types and terminology used here adhere closely to that description.

The abstract type `json_value` represents a JSON *value*. The dynamic type of a polymorphic instance of this class will be one of these extended types:

`json_integer` stores a JSON *number* without fractional part (P)

`json_real` stores a JSON *number* with fractional part (P)

`json_string` stores a JSON *string* (P)

`json_boolean` stores a logical for the JSON literals `true` and `false` (P)

`json_null` represents the JSON literal `null` (P)

`json_object` stores a JSON *object* (S)

`json_array` stores a JSON *array* (S)

The primitive types (P) have a public component `%value` that stores the corresponding value (except for `json_null`). The content of the structure types (S) are accessed via iterator objects. For `json_object` values:

```
type(json_object), target  :: value
type(json_object_iterator) :: iter
iter = json_object_iterator(value)
do while (.not.iter%at_end()) ! order of object members is insignificant
  ! iter%name() is the name of the member
  ! iter%value() is a class(json_value) pointer to the value of the member
  call iter%next
end do
```

For `json_array` values:

```
type(json_array), target  :: value
type(json_array_iterator) :: iter
iter = json_array_iterator(value)
do while (.not.iter%at_end()) ! order of array elements *is* significant
  ! iter%value() is a class(json_value) pointer to the value of the element
  call iter%next
end do
```

The following subroutines allocate and define an allocatable `class(json_value)` variable with JSON text read from a string or logical unit opened for unformatted stream input.

```
call json_from_string(string, value, stat, errmsg)
call json_from_stream(unit,   value, stat, errmsg)
  character(*), intent(in) :: string
  integer, intent(in) :: unit
  class(json_value), allocatable, intent(out) :: value
  integer, intent(out) :: stat
  character(:), allocatable, intent(out) :: errmsg
```

The argument `stat` returns a nonzero value if an error occurs, and in that case `errmsg` is assigned an explanatory error message.

3.2 Examples

Here are some examples that use `json_from_string`. Examples using `json_from_stream` would be essentially the same. Note that the `stop` statements identify things that should not occur.

Reading primitive JSON values:

```
use json

class(json_value), allocatable :: val
character(:), allocatable :: errmsg
integer :: stat

call json_from_string('42', val, stat, errmsg)
select type (val)
type is (json_integer)
  if (val%value /= 42) stop 1
class default
  stop 2
end select

call json_from_string('"foo"', val, stat, errmsg)
select type (val)
```

```
type is (json_string)
  if (val%value /= 'foo') stop 3
class default
  stop 4
end select

call json_from_string('false', val, stat, errmsg)
if (stat /= 0) stop 51
select type (val)
type is (json_boolean)
  if (val%value) stop 5
class default
  stop 6
end select

call json_from_string('null', val, stat, errmsg)
select type (val)
type is (json_null)
class default
  stop 7
end select
```

Reading a JSON array value and iterating through its elements:

```
use json

class(json_value), allocatable :: val
type(json_array_iterator) :: iter
character(:), allocatable :: errmsg
integer :: stat, n

call json_from_string('[42,"foo",false,null]', val, stat, errmsg)

select type (val)
type is (json_array)
  n = 0
  iter = json_array_iterator(val)
  do while (.not.iter%at_end())
    n = n + 1
    select type (ival => iter%value())
    type is (json_integer)
      if (n /= 1) stop 1
      if (ival%value /= 42) stop 2
    type is (json_string)
      if (n /= 2) stop 3
      if (ival%value /= 'foo') stop 4
    type is (json_boolean)
      if (n /= 4) stop 5
      if (ival%value) stop 6
    type is (json_null)
      if (n /= 5) stop 7
    class default
      stop 8
    end select
    call iter%next
  end do
class default
  stop 9
```

```
end select
```

Reading a JSON object value and iterating through its members:

```
use json

class(json_value), allocatable :: val
type(json_object_iterator) :: iter
character(:), allocatable :: errmsg
integer :: stat

call json_from_string('{"a":42,"b":"foo","c":false}', val, stat, errmsg)

select type (val)
type is (json_object)
  iter = json_object_iterator(val)
  do while (.not.iter%at_end())
    select type (ival => iter%value())
    type is (json_integer)
      if (iter%name() /= 'a') stop 1
      if (ival%value /= 42) stop 2
    type is (json_string)
      if (iter%name() /= 'b') stop 3
      if (ival%value /= 'foo') stop 4
    type is (json_boolean)
      if (iter%name() /= 'y') stop 6
      if (ival%value) stop 6
    class default
      stop 7
    end select
    call iter%next
  end do
class default
  stop 8
end select
```

Error handling with invalid JSON:

```
use json

class(json_value), allocatable :: val
integer :: stat
character(:), allocatable :: errmsg

call json_from_string('[1,2,foo,3]', val, stat, errmsg)
if (stat == 0) stop 1 ! should have been an error
write(*,*) errmsg
```

This produces this error output when run:

```
lexical error: invalid string in json text.
                                 [1,2,foo,3]
                     (right here) ------^
```