jRead - an in-place JSON element reader

tonywilk, 12 Mar 2015 CPOL

4.72 (6 votes)

Not 'Just Another Parser', this reads elements from JSON simply and without memory overhead in C

- **Download jRead_1v5-noexe.zip - 80.4 KB**
- **Download jRead_1v5.zip - 80.4 KB**

Introduction

jRead is a simple in-place JSON element reader, it maintains the input JSON as unaltered text and allows queries to be made on it directly. Written in C, jRead is small, fast and does not allocate any memory - making it an ideal choice for embedded applications or those cases where all you want is to read a few values from a chunk of JSON without having to learn a large API and have your program generate lots of interlinked structures to represent the JSON.

Basically the API is a single function call:

```
jRead( pJsonText, pQueryString, pOutputStruct );
```

If you are really adventurous, there are two more functions and a few optional 'helper functions'

Background

In many cases, handling JSON in C or C++ is a real pain. I've looked at several C++ JSON parsers and they all have a learning curve and have to use structures or classes to represent nodes of the JSON.

Furthermore, exclamations like "All I want is a few values out of this load of JSON!", "How the ???? do I use this API for such a simple job ?", "I don't care about writing a JSON reply

- that's the EASY bit !" and "my embedded target isn't made of memory!" are all reasons why I ended up writing jRead.

Points of Interest

The code is writted in pure C, it's all in `jRead.c` and `jRead.h`, it has no dependancies so it should compile on any system for any target. It does not malloc() any memory, it only needs some stack for the function recursion.

It does not so much parse the JSON as *traverse* it - looking for the element you are interested in, It can be used to extract terminal values (strings, numbers etc.) as well as whole objects or arrays. It makes it easy to read JSON and put the values into native C variables, arrays or structures according to your application.

The code also comes with `main.c` which contains a whole bunch of examples and runs as a command-line utility to query JSON files. Project files are included for VisualStudio 10 and a command-line .exe for Windows.

Examples of what jRead does

Assume we have a simple JSON string we have read into some buffer `pJson` like:

```
{
    "astring":"This is a string",
    "anumber":42,
    "myarray":[ "zero", 1, {"description":"element 2"}, null ],
    "yesno":true,
    "PI":"3.1415926",
    "foo":null
}
```

we can get the value of any element using jRead e.g.

```
struct jReadElement result;
jRead( pJson, "{'astring'", &result );
```

in this case the result would be:

```
result.dataType= JREAD_STRING
result.elements= 1
result.byteLen=  16
```

```
result.pValue -> "This is a string"
```

Note that nothing is copied, the result structure simply gives you a pointer to the start of the element and its length.

The Query String defines how to traverse the JSON to end up with a value which is returned. Elements of a query can be as follows:

```
"{'keyname'"          Object element "keyname", returns value of that key
"{NUMBER"             Object element[NUMBER], returns keyname of that element
"[NUMBER"             Array element[NUMBER], returns value from array
```

Thinking with your Javascript head on, it is fairly obvious that:

```
jRead( pJson, "{'myarray'", &result );
```

would return:

```
result.dataType= JREAD_ARRAY
result.elements= 4
result.byteLen=  46
result.pValue -> [ "zero", 1, {"description":"element 2"}, null ]
```

and stringing query elements lets us retrieve the value in the enclosed object like this:

```
jRead( pJson, "{'myarray' [2 {'description'", &result );
```

giving us

```
result.pValue -> "element 2"
```

You may have noticed that the query string uses 'single quotes' to enclose key names instead of JSON-required"double quotes" - this is just to make it easier to type in a query string (there is an option to change this). Having to use double quotes just ends up messy e.g.

```
jRead( pJson, "{\"myarray\"[2{\"description\"", &result );      \\ ugh!
```

The helper functions make it easy to extract single values into C variables like:

```
jRead_string( pJson, "{'astring'", destString, MAXLEN );
my_int= jRead_int( pJson, "{'myarray'[1" );
my_long= jRead_long( pJson, "{'anumber'" );
my_double= jRead_double( pJson, "{'PI'[3" );
```

Note that the 'int' helper does some 'type coersion' - since everything is a string anyway, calling `jReadInt()` will always return a value: it will return 42 for the JSON 42 or "42" and returns zero for "foo". It also returns 1 for true and 0 for false or null.

Advanced Usage

Assuming we have some sensible JSON (not a contriveed mish-mash like the above example), you'll probably have arrays of stuff, then you'll say that indexing thru an array is a pain 'cos you have to build a query string.

Ah ha! I say, you can use a `Query Parameter!`

More likely you'll have some JSON with the bit you want embedded in it like:

```
{
  "Company": "The Most Excellent Example Company",
  "Address": "Planet Earth",
  "Numbers":[
             { "Name":"Fred",   "Ident":12345 },
             { "Name":"Jim",    "Ident":"87654" },
             { "Name":"Zaphod", "Ident":"0777621" }
           ]
}
```

you could extract the names and numbers by:

```
#define NAMELEN 32
struct NamesAndNumbers{      // our application wants the JSON "Numbers" array data
    char Name[NAMELEN];      // in an array of these structures
    long Number;
};
...
struct NamesAndNumbers people[42];
struct jReadElement element;
int i;
  jRead( pJson, "{'Numbers'", &element );    // we expect "Numbers" to be an array
  if( element.dataType == JREAD_ARRAY )
```

```
{
    for( i=0; i<element.elements; i++ )    // loop for no. of elements in JSON
    {
        jRead_string( pJson, "{'Numbers'[*{'Name'", people[i].Name, NAMELEN, &i );
        people[i].Number= jRead_long( pJson, "{'Numbers'[*{'Ident'", &i );
    }
}
```

... in other words, you can supply a pointer to an `int` (or `int[]`) whose value is used in place of the `*` marker in the query string (a bit like parameters in prinf()). See `main.c runExamples()` for use of an index array.

How the code works

The input string containing the JSON must be `'\0'` terminated, which gives us backstop for searching, likewise the query string is `'\0'` terminated.

In contract to a usual parser, this traverses the JSON while traversing the query string, simply skipping elements while tracking the hierarchy of objects and arrays.

The routine extracts a token from both JSON and Query, there are two valid outcomes of this:

1. There is nothing left in the query
2. The tokens match

If there is nothing left in the query, we want the result to be the remaining JSON at this point, i.e. we return the whole of the JSON value (if the query is initially empty, the return is the whole of the input JSON),

If the tokens match (e.g. they are both `'{'` which is token `JREAD_OBJECT`), then we expect the query string to have an additional specifier:

1. an index if it's an array (or we want a 'key' from an object)
2. a key value if it's an object

At this point we need to step thru the JSON until we find the element of interest, in the case of an object, we expect elements to be of the form:

```
"key string" JREAD_COLON <value> JREAD_COMMA | JREAD_EOBJECT
```

if the key string is the one we want (it matches the query key), then we recursivly call `jReadParam()` pointing to the start of the expected <value> in the JSON and a pointer to whatever remains of the query string.

If this key string is not the one we want, we simply call `jRead()` *with an empty query string* to traverse the whole of the JSON value that we're not interested in.

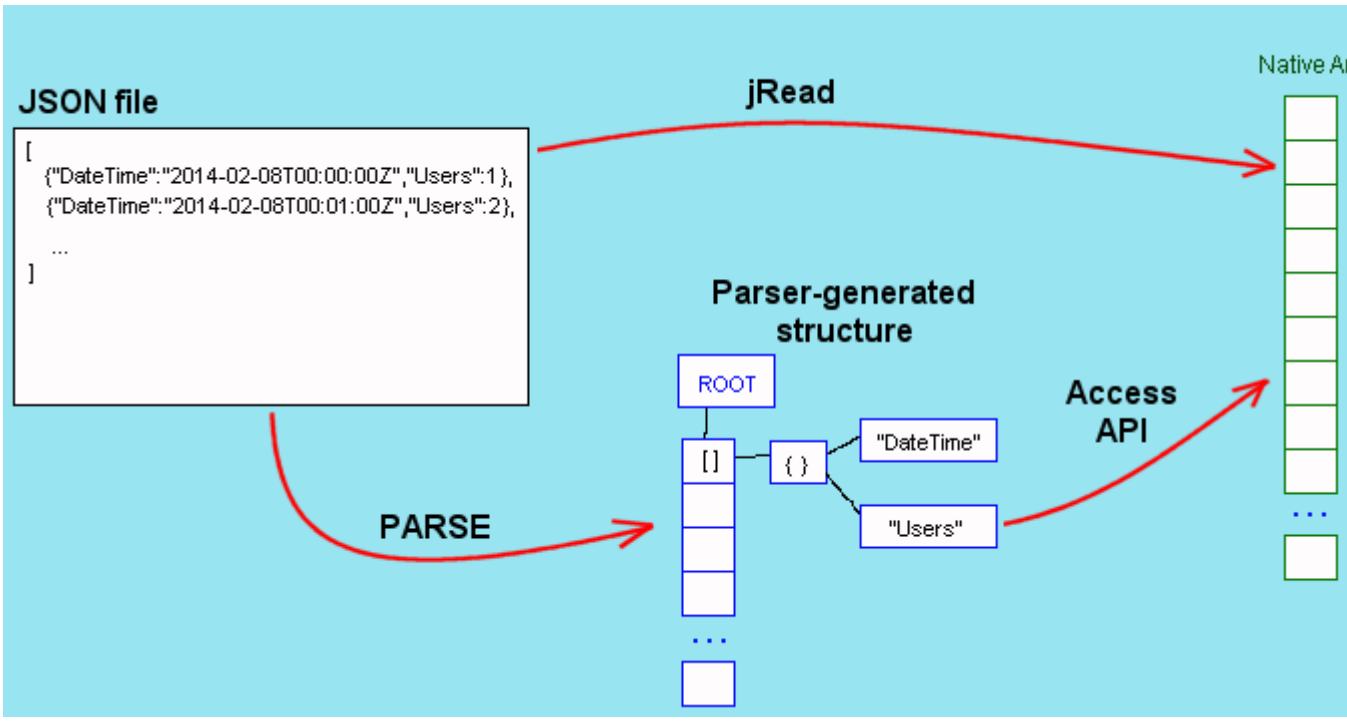
At the end of the query string we may have a 'terminal value' (number, string, bool etc.) or we may have an object or array. If it's terminal, we just return that single value element. Otherwise we want to return the number of elements in the object/array.

The two separate routines jReadCountObject() and jReadCountArray() are used to step through objects and arrays and return their length and element counts.

'Traversing' vs. 'Parsing'

Although, strictly speaking, '*to parse*' means to resolve some input into its component parts, a JSON *parser* is usually expected to create some data structure representing the input JSON. jRead also has to 'parse' the input but may be better described as traversing the JSON to extract an element.

For an application to make use of any JSON data, that data must end up as some native data type (or structure/object). A parser does not usually do this, the parsed JSON ends up in a data structure which is accessed bu functions or methods forming an API whereas the intention for jRead is to directly fill some application-specific data structures.



The example here is to read integer values ("Users") from a JSON array of objects into a C integer array.

Speed

Well, it depends...

Since jRead tranverses the JSON looking for an element to return, it can do this pretty quickly - however, it has to do this for every value you want. Also, an element near the front is found fast (the remainder of the input JSON is not even looked at) whereas for an element at the end of the input we have to traverse the whole thing.

In the 'Advanced' Query Parameter example above, we further confused the 'speed' issue by getting the "Numbers" element from the object and then performing further queries on that (i.e. we start a jRead query pointing at the array embedded in the JSON) - so this saves having to traverse the enclosing object again and again.

I have done some speed tests and there is a simple test built into the command-line program in the latest version. Using this, my 3.4GHz i7 in a DOS box under Windows 7 managed 770,000 queries/sec (timed by leaving it in a loop to do a few million) which is about 1.3uS/query. Further messing about with longer JSON files allowed me to estimate c. 50nS per JSON element traversed. That sounds pretty fast !

Ah, not so fast... If you have a JSON array containing 10,000 objects like:

```
[ {"DateTime":"2014-02-08T00:00:00Z","Users":1},
  ...
  {"DateTime":"2014-02-14T22:39:00Z","Users":10000} ]
```

then it takes around 1.4mS to get the last one. However, if you called jRead 10,000 times to read every entry, it takes an average of 0.74mS per entry - a total of 7.4 *seconds* to do the whole lot.

Which is why I added the 'step' functions - see below.

Adding the Array Step function

Because the jRead performance on large arrays was so poor, I added another function:

```
jReadArrayStep( char *pJsonArray, struct jsonElement *result )
```

Realising that the whole point of jRead is simply to extract values from JSON into whatever nice C variables you have and that jRead works by traversing the JSON - it was a obvious to add this function.

`jReadArrayStep()` simply extracts one element from an array, returns that in the `jsonElement` and returns a pointer to the remainder of the array.

```
// identify the whole JSON element
jRead( (char *)json.data, "", &arrayElement );
if( arrayElement.dataType == JREAD_ARRAY )          // make sure we have an array
{
    int step;
    pJsonArray= (char *)arrayElement.pValue;
    for( step=0; step<arrayElement.elements; step++ )   // step thru all array elements
    {
        pJsonArray= jReadArrayStep( pJsonArray, &objectElement );
        if( objectElement.dataType == JREAD_OBJECT )    // we expect an object
        {
            // query the object and save in our application array
            UserCounts[step]= jRead_int( (char *)objectElement.pValue, "{'Users'",
NULL  );
        }else
        {
            printf("Array element wasn't an object!\n");
        }
    }
}
```

Now we're getting somewhere: instead of 7.4 seconds to do separate indexing queries, the use of jReadArrayStep() can read all 10,000 values in a mere 3.1mS. (using the above code there is an additional 1.5mS for identifying the whole of the input to get `arrayElement` - a total of 4.8mS).

You could add a similar 'step' function for JSON Objects and return both Key and Value. I havn't implemented this because you would still have to string match the returned Key in order to use the Value and usually JSON objects don't have large numbers of keys.

Comparison with a parser

I have compared jRead to my heavily-modified version of simpleJSON in C++ (original project was:https://github.com/MJPA/SimpleJSON)

Parsing the JSON 10,000 object array with SimpleJSON took 48.4mS, plus another 0.4mS to read 10,000 values into an int array, a total of 48.8mS and used a few Mb for the structures.

Comparing this to 4.8mS with no memory overhead, jRead looks like a clear winner in this case.

Of course, a few mS here or there may not mean much - but this is on a fast 3.4GHz i7 machine, running on an embedded device could be 100's of times slower and the differences very significant (On an old Sony 1.2GHz Pentium 'M' laptop, simpleJson took 282.2mS vs. 19.1mS for jRead).

Use Cases for jRead

If you have an embedded system or otherwise can't afford to burn memory then jRead has a distinct advantage. If your input JSON is small-ish (a few Kb) - usually the case for passing parameters between communicating applications - then jRead saves the memory and time for parsing. If you only want a subset of a whole bunch of JSON then you can use jRead to locate the element you are interested in and then either query that - or even use a parser on that element.

Translating large arrays of objects into some native format is pretty fast if you use the array step function since it only traverses the source once to get all your values where you want 'em.

Remember that in C (or C++) the structure defined in a JSON file does not map to the language well at all, a parser has to build arrays/structs/objects to represent each node in the JSON. Once the parser has done it's job, you still have to traverse the structure to actually get at a terminal value.

The last winning point for jRead is its simplicity: it just compiles and runs, you don't have to learn some API or figure out how to traverse a bunch of class objects after you've called yetAnotherParser::Parse()

If you have to deal with a lot of JSON and don't want to translate the data into some application storage of your own but want to continually access values, then jRead is at a disadvantage when it has to travers the source every time. If you want to read/modify/write an amount of JSON, then a parser would be a better bet.

Conclusion

Hope this is of some use to people, it's free to use, copy, rewrite or whatever. Have a look at the code, there's lots of comment, info and examples.

If you do use it and happen to meet me in a pub that sells good beer... I'll have a pint of Tim Taylors - thanks!

TonyWilk

License

About the Author

tonywilk

Founder

United Kingdom

Developer in just about anything from 6502 machine code thru C, C++ and now things like PHP and javascript. Used to develop hardware and still dabble with electronics and ham radio when I'm not letting off pyrotechnics, shooting or flying my VPM M16 gyroplane.