Welcome to libjson 6. libjson is the fastest and more customizable json library for C and C++. It has a special interface for C++, to allow C++ programmers to embed libjson within their program, and because many of libjson's functions are inlined, they will get a major optimization boost from the compiler.

It also has a C-style interface for other programming languages and for sharing libraries. This document will explain building and optimizing libjson to fit your needs, the C interface, and the C++ specialized interface. Each has their own part in this document.

If you are upgrading from libjson 5 or earlier, this new library has a different interface, so your programs will break if you try and plug and play, but it should not be difficult to convert them, as all of the functions are still there, they just have more standardized names.

You should look at the changelog at the end of the document each time you upgrade, as things may have been added or fixed that you are using.

## About libjson

libjson is a high speed complete JSON library, including parsers, writers, builders, formatters… It is also highly customizable. It may seem strange that libjson is considered a very lightweight library, but the documentation is 200 pages long and there are dozens of source files, but the fact is that very little of the library gets complied at any given time. The JSONOptions.h file sets options for which parts are compiled and how they are. This document will tell you if function behave differently or are unavailable under certain options.

libjson was written with performance in mind. It is fast, and uses very little memory. Low level memory manipulation allows libjson to do some speed tricks that less fine-tuned libraries don't do. The loads of preprocessor options means that there are few conditions at run-time, allowing it to focus as much energy on the task at hand. The options allow you to optimize libjson for your program. See the Optimizing libjson section.

I also gave it more customizability than any other library, including mutex managers, garbage collection, memory control, and unicode support, all optional of course, and not on by default. There is a clean C interface, with all methods using C - style names and all prefixed with json_. There is also a very intuitive C++ interface complete with iterators, and a very STL - style interface. These two libraries share some code, but also have version specific code.

Please report any bugs or feature requests here: http://sourceforge.net/tracker/?group_id=314112

# Building libjson 7

# C interface 37

# C++ interface                              108

# Optimizing libjson                                                  172

# Changelog                                                           174

# License                                                             177

# Building libjson

libjson is designed for customizability by experts, but also ease of use by novices.  Most users simply need to make the library.  If you are planning on embedding it in your C++ program, then all you have to do is add libjson's source to your project.  libjson does not use anything other than built-in C++ and any compiler should compile it.

    cd libjson
    make [debug/small]

There are a couple options for making libjson:

|  | **make** | **make debug** | **make small** |
|---|---|---|---|
| Output | libjson.a | libjson_dbg.a | libjson.a |
| Library Options | JSONOptions.h | JSONOptions.h, JSON_SAFE, and JSON_DEBUG | JSONOptions.h and JSON_LESS_MEMORY |
| Compile Options | -O3, -ffast-math, -fexpensive-optimizations | -g -Wall | -Os, -ffast-math |

To compile libjson, you need a functional C++ compiler.  GCC is freeware.

## JSONOptions.h

In the source code you will find a file called JSONOptions.h, which is used for configuring libjson's build.  All of the options are in there and it's recommended that you leave them, just comment out what you don't need.  There is a brief description of each option in the file, as well as a more extensive explanation further down in this document.

I chose to use a file for the options in libjson instead of forcing the user to know compile options and change them in the project and make files.  I think this is easier.

It is recommended that you build and run the Unit Test before compiling for the library or using it in C++.  As there are billions of combinations of options, not all of them have been tested together.  If the build failed, please take out a ticket on sourceforge: http://sourceforge.net/tracker/?group_id=314112 or email the author at ninja9578@yahoo.com.  Your issue will be quickly fixed and your options added to the quarter of a million unit tests that are already run, that way regression of the bug is impossible.

If you email from your sourceforge account, be sure that you're able to receive replies, a lot of you who try and contact me don't so I can't respond.

**JSON_LIBRARY**

This option exposes only the C-style interface, which is used for static and dynamic library compilation so that it can be shared by multiple programs, or used by non-C++ programming languages. This C interface is a little simpler so that it is compatible with most programming languages primitives, but the overall functionality remains the same.

**Recommended for**

Sharing the library among multiple programs, dynamically loading the library, or using it in a programming language other than C++.

**On by default**

Yes

**JSON_DEBUG**

This option is used to create a debugging version of the library. It does a lot of checks not only of the json, but also of the library itself. This is recommended only while writing your software, but not for release, as it is a little bit slower.

This option exposes libjson::registerDebugCallback in C++ and json_register_debug_callback in C, which allows you to get feedback from libjson if something goes wrong. This will not protect against errors, only let you know about them before it crashes. If you want to catch errors and handle them gracefully, then turn on JSON_SAFE as well. You can also turn on JSON_STDERROR, which hides the callbacks and prints problems to stderr.

This option also exposes the JSONNode::dump function in C++, which allows you inspect the internal workings of libjson, which can be useful for debugging. I chose to omit this from the C interface because it is mostly used by the author and maintainers.

**Recommended for**

Writing and debugging software, not for release software

**On by default**

No

**JSON_SAFE**

This option is also used for debugging, but can also be useful if you think that your program might receive json that is not properly formatted.  If libjson encounters something it it considers an error, then it will tell you through the callback or stderr, depending on the other options. then handle it gracefully.  This option is required for validation.

**Recommended for**

Programs that might receive incorrect json, or if you need to validate json.  If your sure that your json will be correct, then there is no need for this option.

**On by default**

Yes

**JSON_STDERROR**

This option is will route all error messages to cerr instead of the callback.  It hides libjson::registerDebugCallback and json_register_debug_callback as they are not needed.  You can see the errors in the console.

**Recommended for**

Debugging for those who do not want to handle error messages in a special way.

**On by default**

No

**JSON_PREPARSE**

This option forces libjson to parse the entire json string at once.  By default, libjson does on-the-fly parsing for speed reasons.  This makes initial parsing much faster and use much less memory, but subsequent reads of it slightly slower.  This options switches those.

**Recommended for**

Programs that use all or most of the json that it receives.  Because json is largely used for communication from client to server, often the server will send more information that needed, so parsing all of it wastes time for no reason.  However, json can also be used for configuration files as well as other things, so in cases like that, it might be preferable to use this option.

**On by default**

No

**JSON_LESS_MEMORY**

This option makes libjson use about 20% less memory on average.  When libjson is done using memory, it doesn't always release all of it, it keeps it allocated so that any future requests to push memory back is much faster.  This option doesn't allow that to happen, it instantly releases all extra memory.

**Recommended for**

Programs that need as much memory as possible.  Possibly for embedded programming, or programs that might end up using billions of nodes and need as much memory as possible.

**On by default**

No

**JSON_UNICODE**

This option makes libjson use wide strings both internally and for the interface.  Because the JSON standard specifies that it must support full UTF8, libjson can not meet this standard without wide characters.  However, because these multi-byte characters are rarely used, this is optional, and off by default.

**Recommended for**

Programs that need the full range of UTF-8.

**Requirements**

JSON_ISO_STRICT is not possible to use with this option, as they contradict each other.

**On by default**

No

**JSON_REF_COUNT**

This option makes libjson's nodes reference count and copy-on-write it's internal structure.  This makes passing by value and copying nodes very fast, but less thread-safe.

**Recommended for**

Programmers that want to copy and pass by value often.  Also by programmers who take JSONNodes just to read them.  It is recommended if you are interacting with your json with multiple threads, that you turn this option off, or use the JSON_MUTEX_CALLBACKs options and use critical sections.

**On by default**

Yes

**JSON_BINARY**

This option turns on the set_binary and get_binary functions. Because the JSON standard offers no way to encode true binary data, libjson (like most json libraries) uses Base64 to encode and decode binary data into text. This allows you to transport binary data between server and client, such as images or files. If you already have a base64 library, then you don't need this, however, this one is highly optimized and will probably be faster than most others.

**Recommended for**

Programs that have to encode and decode binary data such as images. It can also be used to obfuscate data.

**On by default**

No

**JSON_ITERATORS**

This option turns on iterator functions for libjson.  erase, find, insert are exposed when this option is on.  In C++ mode you'll find iterator functions to be very STL-like, and there is even support for iterators in the library, but more restrictive.

**Recommended for**

Programmers familiar and comfortable with STL iterators or need advanced manipulation methods.

**On by default**

No

**JSON_MEMORY_CALLBACKS**

This option exposes functions to register callbacks for allocating, resizing, and freeing memory. Because libjson is designed for speed, it is feasible that some users would like to further add speed by having the library utilize a memory pool.  With this option turned on, the default behavior is still done internally unless a callback is registered

**Recommended for**

Programs that want to use custom memory handling functionality, such as memory pools and garbage collection.

**On by default**

No

**JSON_MEMORY_MANAGE**

This option exposes functions to bulk delete all memory that libjson has allocated for you.  This includes strings and nodes.  Usually, you are required to keep track of all of the nodes and strings that libjson creates, but with this option you don't have to.

**Recommended for**

Programs that use lots of strings and nodes and want to de bulk cleanups or memory managers.

**On by default**

No

**JSON_MUTEX_CALLBACKS**

This option exposes functions to register callbacks to lock and unlock mutexs and functions to lock and unlock JSONNodes and all of it's children.  This does not prevent other threads from accessing the node, but will prevent them from locking it. Because of libjson's extremely complex internal working and reference counting, it is much easier for the end programmer to allow libjson to manage your mutexs because of reference counting and manipulating trees, libjson automatically tracks mutex controls for you, so you only ever lock what you need to.

**Recommended for**

Programs that require libjson be used in critical sections.

**On by default**

No

**JSON_MUTEX_MANAGE**

This option lets you set mutexes and forget them, libjson will not only keep track of the mutex, but also keep a count of how many nodes are using it, and delete it when there are no more references. This changes the registerMutexCallbacks and json_register_mutex_callbacks to require a third deleter callback as well.

**Recommended for**

Programs that require libjson be used in critical sections and handle it's own mutexes.

**Requirements**

JSON_MUTEX_CALLBACKS is required to also be defined, if it is not, the build will fail.

**On by default**

No

**JSON_ISO_STRICT**

This option strips libjson of anything that is not ISO-C++ standard including long long and a few other datatypes and functions.

**Recommended for**

Programmers that are restricted by company standards that require ISO-C++ code.

**Requirements**

JSON_UNICODE is not possible to use with this option, as they contradict each other.

**On by default**

No

**JSON_NO_C_CONSTS**

This option removes the const qualifier for the function declarations.  The functions do not behave any differently, they simply have a slightly different interface.

**Recommended for**

Programmers who are using legacy C compilers or coding guidelines

**On by default**

No

**JSON_WRITER**

This option exposes the write and write_formatted functionality.  Without this option, libjson is read-only, this allows you to write json as well.

**Recommended for**

Programs that need to write json, such as two way communication or recording a config file after it has changed.

**On by default**

Yes

**JSON_NEWLINE**

This option is used for writing formatted json using specialized newlines.  By default, libjson will use the UNIX newline character.  Simply defining this option is not adequate, you must define it to something.  Whatever it's defined to will be used.  The JSONOptions.h file included with libjson shows you how, as it defined is as a carriage return followed by a newline character, this is standard on Windows or MS-DOS.  You may also want to set it to </br> (the new line tag for HTML.)

Even if you are using unicode, write the value of the newline in regular quotes, libjson will automatically make ii unicode for you (in preprocessor.)

**Recommended for**

Programs that write formatted JSON that require special newline characters.

**On by default**

No

**JSON_INDENT**

This option is used for writing formatted json using specialized indents.  By default, libjson will use the ASCII tab character.  Simply defining this option is not adequate, you must define it to something. Whatever it's defined to will be used.  The JSONOptions.h file included with libjson shows you how, you will most commonly change it to spaces, but HTML might also be common.

Even if you are using unicode, write the value of the newline in regular quotes, libjson will automatically make ii unicode for you (in preprocessor.)

**Recommended for**

Programs that write formatted JSON that spaces to indent instead of tabs.

**On by default**

No

**JSON_ESCAPE_WRITES**

This option is used for writing json where the special characters have been escaped.  This is on by default, because without it, the json no longer meets the standards.  If it's off special characters like newlines, tabs, and special unicode characters are simply place in the json as they appear.  This may cause problems with other JSON engines, and it will not pass validation, so it is recommended that you leave it on.

**Recommended for**

Programs that write JSON and want to adhere to the JSON standard, or communicate between other json engines.

**On by default**

Yes

**JSON_COMMENTS**

This option tells libjson to store and write comments.  libjson always supports parsing json that has comments in it as it simply ignores them, but with this option it keeps the comments and allows you to insert further comments.

libjson will automatically determine if comments are multiline or single line and write them accordingly.

**Recommended for**

Programs that write complex JSON that might be read by human eyes.  Comments make JSON easier to understand and alter.

**On by default**

No

**JSON_WRITE_BASH_COMMENTS**

This option tells libjson to use bash comments to output json with comments. Bash comments are the # character. This also disables multi-line comments in the C-style (/* */). Instead it will simply put multiple lines, all with leading # characters.

**Recommended for**

Programs that require bash comments, perhaps for DOxygen documentation.

**On by default**

No

**JSON_WRITE_SINGLE_LINE_COMMENTS**

This option tells libjson to not use the C-style multi-line comment, as some JSON libraries don't support them.  This will write multi-line comments as a series of // comments instead.

**Recommended for**

Programs that need to output JSON that libraries that don't support multiline comments need to read.

**On by default**

No

**JSON_VALIDATE**

This option exposes validation functions.  For programs that might get invalid json, validation might be required before anything is done.  This option requires JSON_SAFE as it uses some of the safety catches to do the validation without crashing.

**Recommended for**

Programs that might receive invalid json.

**Requirements**

JSON_SAFE is required to also be defined, if it is not, the build will fail.

**On by default**

No

**JSON_CASE_INSENSITIVE_FUNCTIONS**

This option exposes case-insensitive functions.  This includes at, get, find...

**Recommended for**

Programs that don't know the cases of the node names

**On by default**

No

**JSON_INDEX_TYPE**

This option changes the type that is used to track the number of children.  Usually unsigned int is used because it's fast and efficient, but there are cases where it's not desired.  For instance, on embedded systems where there is little memory, one might choose to use a short or even a char.  Or on huge 64-bit systems where the number of nodes may go outside the range of an unsigned int.

**Recommended for**

Either very large or very small systems

**On by default**

No

**JSON_BOOL_TYPE**

This option changes the bool type in the C interface.  This allows you to change the interface to fit the language that you are working with.  If this option is not on, then the bool type is an int, but you can make it anything.

**Recommended for**

Programming languages other than C-based ones

**On by default**

No

**JSON_STRING_HEADER**

This option changes the string type that libjson uses for both it's interface and internals. This can be very useful for working with other libraries. For instance, developers using Qt or wxWigdets. Both are very common and have their own string class. Both should be able to be dropped right in without much problem. The class must be named or typedefed json_string.

**Recommended for**

Making interface between libjson and other libraries easier.

**On by default**

No

**JSON_UNIT_TEST**

This option is used to maintain and debug the libjson.  It makes all private members and functions public so that tests can do checks of the inner workings of libjson.  This should not be turned on by end users.

**Recommended for**

libjson maintainers only.

**On by default**

No

# C interface

libjson has an interface that uses standard C types and a standard C interface.  This interface doesn't have to be used in just C, it can be used in Basic, C, C++ and any other language that supports the C interface (most do.)  All methods in libjson begin with "json_" to make sure that the names don't collide with other libraries or any of your methods.

## JSONNODE types

| Function | Description |
|----------|-------------|
| JSON_NULL | Blank node or "null" |
| JSON_STRING | A string |
| JSON_NUMBER | A floating point number |
| JSON_BOOL | A boolean "true" or "false" |
| JSON_ARRAY | An array of JSONNodes |
| JSON_NODE | A complex JSONNode structure |

## libjson types

| Function | Description |
|----------|-------------|
| JSONNODE * | Opaque pointer to a node within libjson |
| JSONNODE_ITERATOR | Random access iterator |
| json_char | Either char or wchar_t, depending on options |
| json_number | Either float or double, depending on options |
| json_index_t | Child node indexing type |
| json_bool_t | Boolean type |
| json_error_callback_t | typedef void (* json_error_callback_t)(const json_char *) |
| json_mutex_callback_t | typedef void (* json_mutex_callback_t)(void *) |
| json_malloc_t | typedef void * (* json_malloc_t)(unsigned long) |
| json_realloc_t | typedef void * (* json_realloc_t)(void *, unsigned long) |
| json_free_t | typedef void (* json_free_t)(void *) |

## JSONNODE functions

| Function | Description |
| --- | --- |
| json_new | Construct JSONNode |
| json_new_a | Constructs a string node |
| json_new_i | Constructs a integer node |
| json_new_f | Constructs a floating point node |
| json_new_b | Constructs a boolean node |
| json_copy | Copy JSONNode content, usually reference counting |
| json_duplicate | Copy JSONNode content, forcing a copy |
| json_delete | Deletes the node |
| json_delete_all | Deletes all allocated nodes |

## Inspector functions

| Function | Description |
| --- | --- |
| json_type | The type of JSONNode it is |
| json_size | The number of child nodes |
| json_empty | Tests if the node has children |
| json_name | The name of the node |
| json_get_comment | The comment attached to the node |
| json_as_string | The string value of the node |
| json_as_int | The integer value of the string |
| json_as_float | The floating point value of the string |
| json_as_bool | The boolean value of the string |
| json_as_node | The node, cast to a JSON_NODE |
| json_as_array | The node, cast to a JSON_ARRAY |
| json_as_binary | The node with it's value converted to binary |
| json_write | Writes the node as JSON text |
| json_write_formatted | Writes the node as readable JSON text |
| json_equal | Compare JSONNode contents |

**Modifier Functions**

| Function | Description |
| --- | --- |
| json_set_a | Sets the contents of the node to a string |
| json_set_i | Sets the contents of the node to an int |
| json_set_f | Sets the contents of the node to a float |
| json_set_b | Sets the contents of the node to a bool |
| json_set_n | Sets the contents of the node to another node |
| json_set_name | Sets the name of the node |
| json_set_comment | Sets the comment attached to the node |
| json_clear | Removes all children |
| json_nullify | Nulls out the node |
| json_swap | Swap the contents of two nodes |
| json_merge | Merges the contents of two or more nodes |
| json_preparse | Completely parses the JSON |
| json_set_binary | Sets the binary value of the node |
| json_cast | Change the node's type |

**Children Access Functions**

| Function | Description |
| --- | --- |
| json_at | Access item by index |
| json_get | Access item by name or index |
| json_get_nocase | Assess item by name, case-insensitive |
| json_reserve | Reserve enough space |
| json_push_back | Adds a child |
| json_pop_back_at | Removes and returns an item by index |
| json_pop_back | Removes and returns an item by name |
| json_pop_back_nocase | Removes and returns an item by name, case-insensitive |
| json_find | Finds a node by name |
| json_find_nocase | Finds a node by name, case-insensitive |
| json_erase | Removes an item |
| json_erase_multi | Remove a set of children |
| json_insert | Adds a child |
| json_insert_multi | Insert a set of children |

**Iterator Functions**

| Function | Description |
| --- | --- |
| json_begin | Return iterator to beginning |
| json_end | Return iterator to end |

**Thread Safety Functions**

| Function | Description |
| --- | --- |
| json_lock | Return iterator to end |
| json_unlock | Return reverse iterator to reverse beginning |
| json_set_mutex | Attaches a mutex to the node |
| json_set_global_mutex | Sets the global mutex |

**JSON Functions**

| Function | Description |
| --- | --- |
| json_parse | Parses json |
| json_strip_white_space | Removes all white space and comment |
| json_validate | Validates json |

**Callback Registration**

| Function | Description |
| --- | --- |
| json_register_debug_callback | Registers error callback |
| json_register_mutex_callbacks | Register mutex callbacks |
| json_register_memory_callbacks | Registers the memory callbacks |

**Cleanup Functions**

| Function | Description |
| --- | --- |
| json_free | Frees memory of a string |
| json_free_all | Frees memory of all strings that libjson had allocated |

**Text Functions**

| Function | Description |
| --- | --- |
| JSON_TEXT | Creates a text string in the right format |

## json_new

JSONNODE * json_new (char mytype);

### New Node

   This function creates an empty node of the specified type.  This would normally be used to start serializing something or writing a configuration file.  You must json_delete the resulting node or attach it to something as a child.

### Option Differences

JSON_MEMORY_MANAGER - Will keep a reference to the resulting node, so it doesn't have to be
   explicitly json_deleted.

### Parameters

mytype
   The enumerated type of the JSONNODE.

### Return Value

Empty node

### Complexity

Constant

## json_new_a

JSONNODE * json_new_a (const json_char * name, const json_char * value);

**New Node**

This function creates a string node with the name and value specified.  You must json_delete the resulting node or attach it to something as a child.

**Option Differences**

JSON_MEMORY_MANAGER - Will keep a reference to the resulting node, so it doesn't have to be
        explicitly json_deleted.
JSON_SAFE - Allows value to be NULL

**Parameters**

name
        The node's name, this may be NULL
value
        The node's value, this should not be NULL

**Return Value**

A new node

**Complexity**

Constant

## json_new_i

JSONNODE * json_new_i(const json_char * name, long value);

### New Node

This function creates a integer node with the name and value specified.  You must json_delete the resulting node or attach it to something as a child.

### Option Differences

JSON_MEMORY_MANAGER - Will keep a reference to the resulting node, so it doesn't have to be explicitly json_deleted.

### Parameters

name
        The node's name, may be NULL
value
        The node's value

### Return Value

A new node

### Complexity

Constant

## json_new_f

JSONNODE * json_new_f(const json_char * name, json_number value);

### New Node

This function creates a floating point node with the name and value specified.  You must json_delete the resulting node or attach it to something as a child.

### Option Differences

JSON_MEMORY_MANAGER - Will keep a reference to the resulting node, so it doesn't have to be explicitly json_deleted.

### Parameters

name
The node's name, may be NULL
value
The node's value

### Return Value

A new node

### Complexity

Constant

## json_new_b

JSONNODE * json_new_b(const json_char * name, int value);

### New Node

This function creates a boolean node with the name and value specified.  You must json_delete the resulting node or attach it to something as a child.

It's important to note that this function takes an int as the value,  Some languages do not have a bool type (including C89.)

### Option Differences

JSON_MEMORY_MANAGER - Will keep a reference to the resulting node, so it doesn't have to be explicitly json_deleted.

### Parameters

name
>    The node's name, may be NULL

value
>    The node's value

### Return Value

A new node

### Complexity

Constant

## json_copy

JSONNODE * json_copy(const JSONNODE * node);

**Copy Node**

       This function copies a JSONNODE and returns the new copy.  With reference counting, this operation is extremely fast.  If you want a literal copy and not a reference count, use json_duplicate.

**Option Differences**

JSON_MEMORY_MANAGER - Will keep a reference to the resulting node, so it doesn't have to be
       explicitly json_deleted.
JSON_REF_COUNT - if this is turned off, the copy constructor will fully duplicate the node

**Parameters**

node
       The node to be copied

**Return Value**

An exact copy of the node parameter

**Complexity**

Constant, unless JSON_REF_COUNT is on, in which case it's the same as json_duplicate.

## json_duplicate

JSONNODE * json_duplicate(const JSONNODE * node);

**Duplicating JSONNODE**

Constructs a JSONNODE object, by copying the contents of JSONNODE.  This is different from the json_copy because it makes a literal copy, not reference counting.

**Option Differences**

None

**Parameters**

node
        The node to be copied

**Return Value**

A JSONNode that is a new copy of the original node.

**Complexity**

Linear on json_size, however, because JSON is a tree structure it's worse case scenario is linear on json_size + json_size of each child recursively.

## json_delete

void json_delete(JSONNODE * node);

**Destruct JSONNODE**

Destructs the JSONNode object and cleans itself up.

**Option Differences**

None

**Parameters**

node
      The node to be deleted

**Return Value**

None

**Complexity**

      Depends on circumstances.  If it's reference count is not one, then it's complexity is constant, if it's the sole owner of it's value, then it becomes linear on JSONNode::size, however, because JSON is a tree structure it's worse case scenario is linear on JSONNode::size + JSONNode::size of each child recursively.

## json_delete_all

void json_delete_all(void);

### Destruct All JSONNODEs

Deletes all nodes that have been allocated by libjson.  This is for bulk delete when you are done with libjson for the time.

### Option Differences

JSON_MEMORY_MANAGER - Is required to use this function

### Parameters

None

### Return Value

None

### Complexity

Complexity of json_delete for each node.

## json_type

char json_type(JSONNODE * node);

**Return type**

Returns the type of the JSONNODE.

**Option Differences**

None

**Parameters**

**Return Value**

The type of the node.

**Complexity**

Constant

## json_size

json_index_t json_size(JSONNODE * node);

**Return size**

Returns the number of children that the node has.  This should be zero for anything other than JSON_ARRAY or JSON_NODE, but this is only guaranteed with the JSON_SAFE option turned on.  This is because casting may or may not purge the children.

**Option Differences**

JSON_SAFE - will guarantee that anything other than JSON_NODE and JSON_ARRAY will return 0.

**Parameters**

None

**Return Value**

The number of children that the node has.

**Complexity**

Constant.

## json_empty

int json_empty(JSONNODE * node);

**Return empty**

Returns whether or not the node has any children.  If the node is not of JSON_NODE or JSON_ARRAY it will invariably return true.  Note that it returns an int, because some languages do not have a bool type.

**Option Differences**

None

**Parameters**

node
>        The node to perform this function on

**Return Value**

Whether or not the node is empty

**Complexity**

Constant.

## json_name

json_char * json_name(JSONNODE * node);

**Return name**

Returns the name of the node.  If there is no name, then it returns a blank string.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

**Return Value**

The name of the string.

**Complexity**

Constant.

## json_get_comment

json_char * json_get_comment(JSONNODE * node);

**Return comment**

Returns the comment attached to the node

**Option Differences**

JSON_COMMENTS - If this option is turned off, this function is not exposed

**Parameters**

node
        The node to perform this function on

**Return Value**

The comment of the node

**Complexity**

Constant.

# json_as_string

json_char * json_as_string(JSONNODE * node);

**Return string value**

Returns the string representation of the node.  This may be different depending on the type of the node

| Function | Description |
|----------|-------------|
| JSON_NULL | "" or "null" depending on how it was created |
| JSON_STRING | The unescaped string value |
| JSON_NUMBER | The number in string form (may be in scientific notation) |
| JSON_BOOL | "true" or "false" |
| JSON_ARRAY | "" |
| JSON_NODE | "" |

Notice that both JSON_NODE and JSON_ARRAY return empty strings.  Use json_write and json_write_formatted to output JSON.

**Option Differences**

None

**Parameters**

node
        The node to perform this function on

**Return Value**

The string representation of the node.

**Complexity**

Constant.

## json_as_int

long json_as_int(JSONNODE * node);

**Return int value**

Returns the integer representation of the node.  This may be different depending on the type of the node

| Function | Description |
| --- | --- |
| JSON_NULL | 0 |
| JSON_STRING | Truncated atof(as_string) |
| JSON_NUMBER | Truncated Value |
| JSON_BOOL | 1 if true, 0 if false |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

If the value is actually a floating point value, then libjson will record an error via it's callback, but will continue on ahead and simply truncate the value.  So 15.9 will be returned as 15.  If JSON_DEBUG is turned on, it will also error if the value of the node is outside the range of the long datatype.

**Option Differences**

None

**Parameters**

node
        The node to perform this function on

**Return Value**

The integer representation of the node.

**Complexity**

Constant.

## json_as_float

json_number json_as_float(JSONNODE * node);

**Return floating point value**

Returns the floating point representation of the node.  This may be different depending on the type of the node

| Function | Description |
| --- | --- |
| JSON_NULL | 0.0 |
| JSON_STRING | atof(as_string) |
| JSON_NUMBER | Value |
| JSON_BOOL | 1.0 if true, 0.0 if false |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

**Option Differences**

None

**Parameters**

node
        The node to perform this function on

**Return Value**

The floating point representation of the node.

**Complexity**

Constant.

## json_as_bool

int json_as_bool(JSONNODE * node);

**Return bool value**

Returns the boolean representation of the node.  This may be different depending on the type of the node

| Function | Description |
|----------|-------------|
| JSON_NULL | FALSE |
| JSON_STRING | Undefined |
| JSON_NUMBER | Value == 0.0 |
| JSON_BOOL | Value |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

**Option Differences**

None

**Parameters**

node
        The node to perform this function on

**Return Value**

The boolean representation of the node.

**Complexity**

Constant.

# json_as_node

JSONNODE * json_as_node(JSONNODE * node);

**Return node value**

Returns the node representation of the node.  For anything other than node and array, it simply returns an empty node.  If the caller is an array, it will convert it to a node.

| Function | Description |
|----------|-------------|
| JSON_NULL | Empty node |
| JSON_STRING | Empty node |
| JSON_NUMBER | Empty node |
| JSON_BOOL | Empty node |
| JSON_ARRAY | Array converted to a node |
| JSON_NODE | A copy of the node |

This command creates a new JSONNODE that has to be deleted or attached to a parent.

**Option Differences**

None

**Parameters**

node
    The node to perform this function on

**Return Value**

The node representation of the node.

**Complexity**

Constant for all except JSON_ARRAYs and JSON_NODEs which are the same as the json_copy function.

## json_as_array

JSONNODE * json_as_array(JSONNODE * node);

**Return array value**

Returns the array representation of the node.  For anything other than node and array, it simply returns an empty array.  If the caller is an node, it will convert it to an array by stripping all of the names of each child.

| Function | Description |
| --- | --- |
| JSON_NULL | Empty node |
| JSON_STRING | Empty node |
| JSON_NUMBER | Empty node |
| JSON_BOOL | Empty node |
| JSON_ARRAY | A copy of the array |
| JSON_NODE | An array of the children |

This command creates a new JSONNODE that has to be deleted or attached to a parent.

**Option Differences**

None

**Parameters**

node
        The node to perform this function on

**Return Value**

The node representation of the node.

**Complexity**

Constant for anything other than a node.  For a JSON_NODE, it is linear with respect to json_size, and JSON_ARRAY which is the same as the json_copy function.

## json_as_binary

void * json_as_binary(JSONNODE * node, unsigned long * size);

**Return binary value**

Returns the binary value that was part of this node.  It returns a byte array which is the original binary data.  It gives you how large the array is by the pass by pointer size parameter.

| Function | Description |
|---|---|
| JSON_NULL | Undefined |
| JSON_STRING | The binary data from the decoded Base64 |
| JSON_NUMBER | Undefined |
| JSON_BOOL | Undefined |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

**Option Differences**

JSON_BINARY - If this option is not on, then this function is not accessible.

**Parameters**

node

      The node to perform this function on

size

      A pointer to an unsigned long, which will tell you how many bytes the data is

**Return Value**

The Base64 decoded binary data

**Complexity**

Linear with respect to the length of the binary data

## json_write

json_char * json_write(JSONNODE * node);

### Write JSON

Returns JSON text, with no white space or comments.  Designed to create JSON that is very small, and therefore, faster to send between servers or write to a disk.  The flipside is that it's nearly impossible to read by human eyes.

Only root nodes (JSON_NODE and JSON_ARRAYs) are meant to be written, all others will return a blank string.

### Option Differences

JSON_WRITER - If this option is not on, then this function is not accessible.
JSON_ESCAPE_WRITES - Escapes special characters

### Parameters

node
       The node to perform this function on

### Return Value

JSON text of the node being written

### Complexity

Linear with respect to the size of the tree underneath the node.

## json_write_formatted

json_char * json_write_formatted(JSONNODE * node);

**Write JSON**

Returns JSON text that has been indented and prettied up so that it can be easily read and modified by humans.

Only root nodes (JSON_NODE and JSON_ARRAYs) are meant to be written, all others will return a blank string.

**Option Differences**

JSON_WRITER - If this option is not on, then this function is not accessible.
JSON_COMMENTS - If this option is not turned on, the no comments are written
JSON_WRITE_BASH_COMMENTS - This option will make libjson write only bash style # comments
JSON_WRITE_SINGLE_LINE_COMMENTS - This option will make libjson not write C-style /* */ comments
JSON_ESCAPE_WRITES - Escapes special characters

**Parameters**

node
        The node to perform this function on

**Return Value**

JSON text of the node being written

**Complexity**

Linear with respect to the size of the tree underneath the node.

## json_equal

int json_equal(JSONNODE * node1, JSONNODE * node2);

### Comparing JSONNODE

Checks if the value held within the nodes are equal.  This ignores things like comments, but for JSON_NODE and JSON_ARRAYs, this is a deep comparison, checking each child too.

### Option Differences

None

### Parameters

node1
    The value to compare to
node2
    Another node to compare to

### Complexity

Constant except for JSON_NODEs and JSON_ARRAYs in which case it's linear with respect to size() and size() of each child recursively.

## json_set_a

void json_set_a(JSONNODE * node, json_char * value);

**Setting the node's value**

Sets the string value of the JSONNODE.

**Option Differences**

JSON_SAFE - Allows value to be NULL

**Parameters**

node
> The node to perform this function on

value
> The new value of the node, shouldn't be NULL unless JSON_SAFE is on

**Return Value**

None

**Complexity**

Constant

## json_set_i

void json_set_i(JSONNODE * node, long value);

**Setting the node's value**

Sets the integer value of the JSONNODE.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

value
> The new value of the node

**Return Value**

None

**Complexity**

Constant

## json_set_f

void json_set_f(JSONNODE * node, json_number value);

**Setting the node's value**

Sets the floating point value of the JSONNODE.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

value
> The new value of the node

**Return Value**

None

**Complexity**

Constant

## json_set_b

void json_set_b(JSONNODE * node, int value);

**Setting the node's value**

Sets the boolean value of the JSONNODE.  The value is an int because of the missing bool datatype in C.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

value
> The new value of the node

**Return Value**

None

**Complexity**

Constant

## json_set_n

void json_set_n(JSONNODE * node, JSONNODE * value);

**Setting the node's value**

Sets the value of the JSONNODE to the value of the other, usually through fast and simple reference counting.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

value
> The new value of the node

**Return Value**

None

**Complexity**

Constant

## json_set_name

void json_set_name(JSONNODE * node, json_char * name_t);

**Setting the node's name**

Sets the name of the JSONNode.

**Option Differences**

JSON_SAFE - name_t can be NULL

**Parameters**

node
        The node to perform this function on
name_t
        The name of the node

**Return Value**

None

**Complexity**

Constant

## json_set_comment

void json_set_comment(JSONNODE * node, json_char * comment_t);

**Setting the node's comment**

Sets the comment that will be associated with the JSONNode.

**Option Differences**

JSON_COMMENTS - this option is required to expose this function
JSON_SAFE - comment_t can be NULL

**Parameters**

node
            The node to perform this function on
comment_t
            The comment attached to the node

**Return Value**

None

**Complexity**

Constant

## json_clear

void json_clear(JSONNODE * node);

**Clearing a node**

Clears all children from the node.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

**Return Value**

None

**Complexity**

Linear with respect to the size of the tree underneath it

## json_swap

void json_swap(JSONNODE * node1, JSONNODE * node2);

**Swaps the contents of nodes**

Swaps the contents of two nodes.  This is very fast because JSONNODE is just a wrapper around an internal structure, so it simply swaps pointers to those structures.

**Option Differences**

None

**Parameters**

node1
>        The node to perform this function on

node2
>        The node to swap values with

**Return Value**

None

**Complexity**

Constant

## json_merge

void json_merge(JSONNODE * node1, JSONNODE * node2);

**Merge the contents of nodes**

It's possible that you may end up with multiple copies of the same node, through duplication and such.  To save space, you might want to merge the internal reference counted structure.  Obviously, this only makes a difference if reference counting is turned on.

libjson will try and do this efficiently, merging with the internal structure that has the highest reference count.

**Option Differences**

JSON_REF_COUNT - this option is required for this to do anything

**Parameters**

node
       The node to perform this function on
node2
       One of the other nodes to merge with.

**Return Value**

None

**Complexity**

Constant

## json_preparse

void json_preparse(JSONNODE * node);

### Preparse the json

libjson's lazy parsing makes parsing JSON that is not entirely used very fast, but sometimes you want to parse it all at once, making the next reads a little faster

### Option Differences

JSON_PREPARSE - this option does this automatically, so it hides this function

### Parameters

node
        The node to perform this function on

### Return Value

None

### Complexity

Linear with respect to the total number of nodes in the JSON

## json_set_binary

void json_set_binary(JSONNODE * node, const void * bin, unsigned long bytes);

**Set binary data**

libjson's built in Base64 encoder will create a JSON_NODE with Base64 encoded binary data, which allows you to send images and files around. libjson's Base64 encoder is faster than most others, so it's recommended that you use this method rather than other libraries.

**Option Differences**

JSON_BINARY - this option is required to use this method and the Base64 method

**Parameters**

node
        The node to perform this function on
bin
        binary data
bytes
        the number of bytes that the binary data is

**Return Value**

None

**Complexity**

Linear with respect to the number of bytes

## json_cast

void json_cast(JSONNODE * node, char type);

**Cast to a different type**

Will change the node to a different type and do any conversions necessary.

**Option Differences**

None

**Parameters**

node
        The node to perform this function on
type
        New type of node

**Return Value**

None

**Complexity**

Constant except for casting a node to array which is linear with respect to json_size.

## json_at

JSONNODE * json_at(JSONNODE * node, json_index_t pos);

**Getting a child**

This will give you a reference to a child node at a specific location.  This is a safe function and will return zero if you go out of bounds.  The returned value is still a child, so do not try and delete the results.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

pos
> The index of the child node

**Return Value**

The child at the desired location.

**Complexity**

Constant

## json_get

JSONNODE * json_get(JSONNODE * node, json_char * name);

**Getting a child**

This will give you a reference to a child node by its name. This is a safe function and will return zero if that child does not exist. The returned value is still a child, so do not try and delete the results.

**Option Differences**

None

**Parameters**

node
>       The node to perform this function on

name
>       The name of the child node

**Return Value**

The child at the desired location.

**Complexity**

Linear with respect to json_size

## json_get_nocase

JSONNODE * json_get_nocase(JSONNODE * node, json_char * name);

### Getting a child

This will give you a reference to a child node by its name.  This is a safe function and will return zero if that child does not exist.  The returned value is still a child, so do not try and delete the results.

### Option Differences

JSON_CASE_INSENSITIVE_FUNCTIONS is required to use this function

### Parameters

node
>       The node to perform this function on
name
>       The name of the child node

### Return Value

The child at the desired location.

### Complexity

Linear with respect to json_size.

## json_reserve

void json_reserve(JSONNODE * node, json_index_t size);

### Reserving space

This function reserves children space, this makes the program faster and use less memory as it doesn't have to keep allocating new memory when it runs out.

### Option Differences

JSON_LESS_MEMORY - Causes this method to do nothing

### Parameters

node
> The node to perform this function on

size
> The size to reserve

### Return Value

None

### Complexity

If there is already some children it's linear with respect to json_size unless the reserved amount is less than the current capacity, in which case it's constant.  If there are no children, then it's constant.

## json_push_back

void json_push_back(JSONNODE * node, JSONNODE * child);

### Adding a child

This function pushes a new child node on the back of the child list.  The child is then managed, so do not try and delete it later.

### Option Differences

None

### Parameters

node
> The node to perform this function on

child
> The node to be added as a child

### Return Value

None

### Complexity

Constant

## json_pop_back_at

JSONNODE * json_pop_back_at(JSONNODE * node, json_index_t pos);

**Getting a child**

This will give remove a JSONNODE from it's parent and return it to you.  Because it's removed from the parent, you must delete it yourself.

**Option Differences**

None

**Parameters**

node
  The node to perform this function on
pos
  The index of the child node

**Return Value**

The child at the desired location by value.

**Complexity**

Linear based on json_size minus pos.  This is because the references of the other children have to be shifted.

## json_pop_back

JSONNODE * json_pop_back(JSONNODE * node, json_char * name);

**Getting a child**

This will give remove a JSONNODE from it's parent and return it to you.  Because it's removed from the parent, you must delete it yourself.

**Option Differences**

None

**Parameters**

node
> The node to perform this function on

pos
> The index of the child node

name
> The name of the child node

**Return Value**

The child at the desired location by value.

**Complexity**

Linear based on json_size.

## json_pop_back_nocase

JSONNODE * json_pop_back_nocase(JSONNODE * node, json_char * name);

**Getting a child**

This will give remove a JSONNODE from it's parent and return it to you.  Because it's removed from the parent, you must delete it yourself.

**Option Differences**

JSON_CASE_INSENSITIVE_FUNCTIONS is required to use this function

**Parameters**

node
        The node to perform this function on
name
        The name of the child node

**Return Value**

The child at the desired location.

**Complexity**

Linear with respect to json_size.

## json_find

JSONNODE_ITERATOR json_find(JSONNODE * node, json_char * name);

**Getting a child**

Searches through the children and finds an iterator to it.  This function returns json_end if the child does not exist.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node
          The node to perform this function on
name
          The name of the child node

**Return Value**

An iterator that points to the child requested, or json_end if it wasn't found.

**Complexity**

Linear based on json_size.

## json_find_nocase

JSONNODE_ITERATOR json_find_nocase(JSONNODE * node, json_char * name);

**Getting a child**

Searches through the children and finds an iterator to it.  This function returns json_end if the child does not exist.

**Option Differences**

JSON_ITERATORS and JSON_CASE_INSENSITIVE_FUNCTIONS must be on to use this function

**Parameters**

node
> The node to perform this function on

name
> The name of the child node

**Return Value**

An iterator that points to the child requested, or json_end if it wasn't found.

**Complexity**

Linear based on json_size.

## json_erase

JSONNODE_ITERATOR json_erase(JSONNODE * node, JSONNODE_ITERATOR pos);

**Removing a child**

Erases a single child and returns the iterator to the next one.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node

        The node to perform this function on

pos

        The position of the node that is to be deleted

**Return Value**

An iterator that points to the child after the deleted one, or json_end

**Complexity**

Linear based on json_size minus where it starts from due to memory shifting.

## json_erase_multi

JSONNODE_ITERATOR json_erase_multi(JSONNODE * node, JSONNODE_ITERATOR start, JSONNODE_ITERATOR end);

**Removing a child**

Erases a single child and returns the iterator to the next one.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node
      The node to perform this function on
start
      The position of starting node to be deleted (inclusively)
end
      The position of ending node to be deleted (inclusively)

**Return Value**

An iterator that points to the child after the deleted one, or json_end

**Complexity**

Linear based on json_size minus where it starts from due to memory shifting.

## json_insert

JSONNODE_ITERATOR json_insert(JSONNODE * node, JSONNODE_ITERATOR pos, JSONNODE * child);

**Adding a child**

These functions place a new child into your node before the node pointed to by the iterator.  This functions copy the nodes, they remain where they are as well, so you must delete it.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node
        The node to perform this function on
pos
        The position to insert before
child
        The node to be inserted

**Return Value**

An iterator that points to the first inserted node

# json_insert_multi

JSONNODE_ITERATOR json_insert_multi(JSONNODE * node, JSONNODE_ITERATOR start, JSONNODE_ITERATOR end);

**Adding a child**

These functions place a new child into your node before the node pointed to by the iterator.  This functions copy the nodes, they remain where they are as well, so you must delete it.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node
> The node to perform this function on

pos
> The position to insert before

start
> Position to start copying (inclusively)

end
> Position to stop copying (inclusively)

**Return Value**

An iterator that points to the first inserted node

**Complexity**

Linear based on json_size minus where it starts from due to memory shifting plus end minus start for the ranged variety.

## json_begin

JSONNODE_ITERATOR json_begin(JSONNODE * node);

**Getting start of children**

This function gets you an iterator pointing to the beginning of the children.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node
> The node to perform this function on

**Return Value**

An iterator that points to the first child

**Complexity**

Constant

## json_end

JSONNODE_ITERATOR json_end(JSONNODE * node);

**Getting end of children**

This function gets you an iterator pointing past the end of the children.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

node
      The node to perform this function on

**Return Value**

An iterator that points past the last child

**Complexity**

Constant

## json_lock

void json_lock(JSONNODE * node, int threadID);

**Lock a mutex**

This function locks the mutex attached to a node, and locks anything sharing the lock in a different thread.  The reason for the threadID parameter is that because libjson passes around structures behind the scenes it might be possible to try and lock the same lock in multiple places in the same critical section.  Counting locks per thread eliminates this problem.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this function

**Parameters**

node
          The node to perform this function on
threadID
          A unique id for this thread

**Return Value**

None

**Complexity**

Constant

## json_unlock

void json_unlock(JSONNODE * node, int threadID);

**Unlock a mutex**

This function unlocks the mutex attached to a node.  If the mutex has been locked multiple times by the same thread, then it will wait until all locks have been released before actually doing the unlock.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this function

**Parameters**

node
　　　　The node to perform this function on
threadID
　　　　A unique id for this thread

**Return Value**

None

**Complexity**

Constant

## json_set_mutex

void json_set_mutex(JSONNODE * node, void * mutex);

### Set a lock

This function attaches a mutual exclusion lock for the node.  It can be locked and unlocked by using the lock and unlock functions,  The reason for this is that because libjson's complex reference counting and internal magic, it is difficult to be sure that you track data correctly, so libjson will pass this mutex around it's internal system and track it for you.

You can pass NULL to this function to unset a mutex.

This method will filter the mutex down to all children as well, and anything that you add to it later.

### Option Differences

JSON_MUTEX_CALLBACKS must be on to use this function
JSON_MUTEX_MANAGER - if this option is on, then libjson will automatically delete this mutex when it runs out of references to it

### Parameters

node
The node to perform this function on
mutex
A pointer to a mutex of any type (POSIX, Windows…)

### Return Value

None

### Complexity

Linear with json_size and json_size of each child recursively.

## json_set_global_mutex

void json_set_global_mutex(void * mutex);

**Register library-wide mutex**

This function sets a global mutex for libjson.  This is not the same as the manager mutex that you are required to register in register_mutex_callbacks.  This mutex is a fallback mutex that gets locked when you attempt to lock a node that has no mutex attached to it.  You may wish to omit assigning mutexes for each node and just let them all use the global one.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this feature

**Parameters**

mutex
          A mutex that libjson will use when it doesn't have any to use

**Return Value**

None

**Complexity**

Constant

## json_parse

JSONNODE * json_parse(const json_char * json);

**Parse JSON text**

      This function parses JSON text and returns you a JSONNode which is the root node of the text that you just passed it.  If bad JSON is sent to this method it may return NULL.

**Option Differences**

JSON_SAFE - a node of type JSON_NULL will be returned if the json was invalid

**Parameters**

json
      JSON text

**Return Value**

The root node of the text

**Complexity**

Linear depending on strlen(json)

## json_strip_white_space

json_char * json_strip_white_space(const json_char * json);

**Parse JSON text**

      This function removes anything that the JSON standard defines as white space, including extra tabs, spaces, formatting, and comments.  This makes this function useful for compressing json that needs to be stored or sent over a network.

**Option Differences**

None

**Parameters**

json
      JSON text

**Return Value**

Valid JSON that is free of all white space

**Complexity**

Linear depending on strlen(json)

## json_validate

JSONNODE * json_validate(const json_char * json);

**Validate JSON text**

This function validates the text by parsing it completely and looking for anything that is malformed.  If bad JSON is sent to this method it will throw a std::invalid_argument exception, otherwise it returns the root node of the text.

**Option Differences**

JSON_VALIDATE and JSON_SAFE must be on to use this feature

**Parameters**

json
JSON text

**Return Value**

The root node of the text

**Complexity**

Linear depending on strlen(json) and the number of nodes within the text.

## json_register_debug_callback
void json_register_debug_callback(json_error_callback_t callback);

### Register error callback

This callback allows libjson to tell you exactly what is going wrong in your software, making debugging and fixing the problem much easier.

### Option Differences

JSON_DEBUG must be on to use this feature

### Parameters

callback
A method that must have this prototype: static void callback (const json_char * message)

### Return Value

None

### Complexity

Constant

# json_register_mutex_callbacks

void json_register_mutex_callbacks(json_mutex_callback_t lock, json_mutex_callback_t unlock, void *
manager_mutex);
void json_register_mutex_callbacks(json_mutex_callback_t lock, json_mutex_callback_t unlock,
json_mutex_callback_t destroy, void * manager_mutex);

**Register mutex callbacks**

This callback allows libjson to lock mutexes that you assign to JSONNode structures. Because
libjson has no idea what kind of mutex you gave it, it can't lock it without the help of a callback. This
method is required to run before any locking can take place.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this feature
JSON_MUTEX_MANAGER must be on to use the second variant

**Parameters**

lock
> A method that locks a mutex and must have this prototype: static void callback (void * mutex).

unlock
> A method that unlocks a mutex and must have this prototype: static void callback (void * mutex).

destroy
> A method that deletes a mutex and must have this prototype: static void callback (void * mutex).
> This parameter is only allowed if JSON_MUTEX_MANAGER is defined, otherwise, you are
> responsible for cleaning up you mutexes.

manager_mutex
> A mutex that libjson will need to use to handle all of the mutexes, as even without
> JSON_MUTEX_MANAGER, some management takes place and libjson has a few critical
> sections when threading. May not be the same as the global mutex.

**Return Value**

None

**Complexity**

Constant

## json_register_memory_callbacks

void json_register_memory_callbacks(json_malloc_t allocer, json_realloc_t reallocer, json_free_t freer);

**Register memory handling**

This function sets callbacks that libjson will use to allocate memory.  This allows you to keep a tight control of memory allocation, or use things like memory pool and such.  These methods are currently not allowed to return null or less than it asked for, as it will cause instability in libjson.

**Option Differences**

JSON_MEMORY_CALLBACKS must be on to use this feature

**Parameters**

allocer
> A method that must have this prototype: static void * alloc(unsigned long bytes)

reallocer
> A method that must have this prototype: static void * realloc(void * buffer, unsigned long bytes)

freer
> A method that must have this prototype: static void dealloc(void * buffer)

**Return Value**

None

**Complexity**

Constant

## json_free

void json_free(void * str);

**Frees text**

      This function removes the memory allocated by various functions that return strings, such as json_as_string, json_as_binary, json_write…

**Option Differences**

None

**Parameters**

str
      Memory that libjson allocated

**Return Value**

None

**Complexity**

Constant unless JSON_MEMORY_MANAGE is defined in which case it's logarithmic depending on the number of strings currently allocated.

## json_free_all

void json_free_all(void);

### Frees text

This function removes all of the memory that libjson has allocated for strings and binary. This allows for bulk frees and garbage collection.

### Option Differences

JSON_MEMORY_MANAGE is required to use this function

### Parameters

None

### Return Value

None

### Complexity

Constant

# JSON_TEXT

## Converts text

This is a marco that you can use to wrap around text that you pass into libjson.  It automatically prepends an L in unicode mode and does nothing when not in unicode mode.  This allows you to use the same code for unicode and non unicode software.

## C++ interface

libjson has an interface specifically designed for C++.  It's designed for the following things: speed, heavy internal optimization, and ease of use.  The interface should be very intuitive to those familiar with STL and uses standard overloaded operators.

Because so many of the functions in JSONNode are inline, the merge between your program and the library at the optimization level allows more optimization that if the library was external.  Your program will run faster and use less memory because of this.

There are two main interfaces here: libjson and JSONNode.  libjson is for parsing and registering callbacks and JSONNode is the worker and the structure that stores and manipulates JSON once it's been loaded.

This part of the document is set up to look like cplusplus.com's reference manual, so that it should be easy to navigate and find everything that you're looking for.

**JSONNode**

JSONNode is the basic class for holding JSON values.  It is a reference-counted, copy-on-write class that adheres to standard C++ practices.  It has a very small footprint, both in memory and on the processor.  It's interface is similar to anything in STL, and should be very intuitive.

There are several types of JSONNodes, but they all share the same class.

**JSONNode types**

| Function | Description |
|---|---|
| JSON_NULL | Blank node or "null" |
| JSON_STRING | A string |
| JSON_NUMBER | A floating point number |
| JSON_BOOL | A boolean "true" or "false" |
| JSON_ARRAY | An array of JSONNodes |
| JSON_NODE | A complex JSONNode structure |

**Member types**

| Function | Description |
|---|---|
| iterator | Random access iterator |
| const_iterator | Constant random access iterator |
| reverse_iterator | A reverse random access iterator |
| reverse_const_iterator | A constant reverse random access iterator |
| json_string | Either std::string or std::wstring, depending on options |
| json_char | Either char or wchar_t, depending on options |
| json_number | Either float or double, depending on options |
| json_index_t | Child node indexing type |
| auto_lock | A scoped locking mechanism for JSONNodes |

## Class functions

| Function | Description |
| --- | --- |
| (constructor) | Construct JSONNode |
| (destructor) | JSONNode destructor |
| duplicate | Copy JSONNode content, forcing a copy |
| operator = | Copy JSONNode content, usually reference counting |

## Inspector functions

| Function | Description |
| --- | --- |
| type | The type of JSONNode it is |
| size | The number of child nodes |
| empty | Tests if the node has children |
| name | The name of the node |
| get_comment | The comment attached to the node |
| as_string | The string value of the node |
| as_int | The integer value of the string |
| as_float | The floating point value of the string |
| as_bool | The boolean value of the string |
| as_node | The node, cast to a JSON_NODE |
| as_array | The node, cast to a JSON_ARRAY |
| as_binary | The node with it's value converted to binary |
| dump | Dumps the inner working of the node into JSON |
| write | Writes the node as JSON text |
| write_formatted | Writes the node as readable JSON text |

## Comparison Functions

| Function | Description |
| --- | --- |
| operator == | Compare JSONNode contents |
| operator != | Compare JSONNode contents |

**Modifier Functions**

| Function | Description |
| --- | --- |
| set_name | Sets the name of the node |
| set_comment | Sets the comment attached to the node |
| clear | Removes all children |
| nullify | Nulls out the node |
| swap | Swap the contents of two nodes |
| merge | Merges the contents of two or more nodes |
| preparse | Completely parses the JSON |
| set_binary | Sets the binary value of the node |
| cast | Change the node's type |

**Children Access Functions**

| Function | Description |
| --- | --- |
| at | Access item by name or index |
| at_nocase | Assess item by name, case-insensitive |
| operator [] | Assess item by name or index |
| reserve | Reserve enough space |
| push_back | Adds a child |
| pop_back | Removes and returns an item by name or index |
| pop_back_nocase | Removes and returns an item by name, case-insensitive |
| find | Finds a node by name |
| find_nocase | Finds a node by name, case-insensitive |
| erase | Removes an item |
| insert | Adds a child |

**Iterator Functions**

| Function | Description |
| --- | --- |
| begin | Return iterator to beginning |
| end | Return iterator to end |
| rbegin | Return reverse iterator to reverse beginning |
| rend | Return reverse iterator to reverse end |

**Thread Safety Functions**

| Function | Description |
| --- | --- |
| set_mutex | Attaches a mutex to the node |
| lock | Return iterator to end |
| unlock | Return reverse iterator to reverse beginning |

**Text Functions**

| Function | Description |
| --- | --- |
| JSON_TEXT | Creates a text string in the right format |

## JSONNode::JSONNode

explicit JSONNode(char mytype = JSON_NODE);
JSONNode(const json_string & name_t, char value_t);
JSONNode(const json_string & name_t, unsigned char value_t);
JSONNode(const json_string & name_t, short value_t);
JSONNode(const json_string & name_t, unsigned short value_t);
JSONNode(const json_string & name_t, int value_t);
JSONNode(const json_string & name_t, unsigned int value_t);
JSONNode(const json_string & name_t, long long value_t);
JSONNode(const json_string & name_t, unsigned long long value_t);
JSONNode(const json_string & name_t, long value_t);
JSONNode(const json_string & name_t, unsigned long value_t);
JSONNode(const json_string & name_t, float value_t);
JSONNode(const json_string & name_t, double value_t);
JSONNode(const json_string & name_t, bool value_t);
JSONNode(const json_string & name_t, const json_string & value_t);
JSONNode(const json_string & name_t, const json_char * value_t);
JSONNode(const JSONNode & orig);

### Construct JSONNode

Constructs a JSONNode object, initializing it's contents depending on which constructor is used:

explicit JSONNode(char mytype = JSON_NODE);
        Default constructor: creates an empty JSONNode of the specified type.
JSONNode(const json_string & name_t, char value_t);
JSONNode(const json_string & name_t, unsigned char value_t);
JSONNode(const json_string & name_t, short value_t);
JSONNode(const json_string & name_t, unsigned short value_t);
JSONNode(const json_string & name_t, int value_t);
JSONNode(const json_string & name_t, unsigned int value_t);
JSONNode(const json_string & name_t, long value_t);
JSONNode(const json_string & name_t, unsigned long value_t);
JSONNode(const json_string & name_t, long long value_t);
JSONNode(const json_string & name_t, unsigned long long value_t);
JSONNode(const json_string & name_t, float value_t);
JSONNode(const json_string & name_t, double value_t);
        Constructs a numeric JSONNode.
JSONNode(const json_string & name_t, bool value_t);
        Constructs a boolean JSONNode.
JSONNode(const json_string & name_t, const json_string & value_t);
JSONNode(const json_string & name_t, const json_char * value_t);
        Constructs a string JSONNode.
JSONNode(const JSONNode & orig);
        Copy constructor, increments the reference counter, will only copy if one of them changes.  To
        make a literal copy, use the duplicate method.

### Option differences

JSON_REF_COUNT - if this is turned off, the copy constructor will fully duplicate the node
JSON_ISO_STRICT - if this option is on, then the long long versions are removed

**Parameters**

mytype
>  The enumerated type of the JSONNode.

name_t
>  The name of the object

value_t
>  The value of the object

orig
>  The original JSONNode that gets copied

**Complexity**

>  Constant with the exception of copying a JSON_NODE or JSON_ARRAY, in which case it is equivalent to JSONNode::duplicate.

# JSONNode::~JSONNode

~JSONNode(void);

**Destruct JSONNode**

Destructs the JSONNode object and cleans itself up.

**Option Differences**

None

**Parameters**

None

**Complexity**

      Depends on circumstances.  If it's reference count is not one, then it's complexity is constant, if it's the sole owner of it's value, then it becomes linear on JSONNode::size, however, because JSON is a tree structure it's worse case scenario is linear on JSONNode::size + JSONNode::size of each child recursively.

# JSONNode::duplicate

JSONNode duplicate(void);

**Duplicating JSONNode**

      Constructs a JSONNode object, by copying the contents of JSONNode. This is different from the copy constructor or assignment operator because it makes a literal copy, not reference counting.

**Option Differences**

None

**Parameters**

None

**Return Value**

A JSONNode that is a new copy of the original node.

**Complexity**

Linear on JSONNode::size, however, because JSON is a tree structure it's worse case scenario is linear on JSONNode::size + JSONNode::size of each child recursively.

# JSONNode::operator =

```
void operator = (char value_t);
void operator = (unsigned char value_t);
void operator = (short value_t);
void operator = (unsigned short value_t);
void operator = (int value_t);
void operator = (unsigned int value_t);
void operator = (long long value_t);
void operator = (unsigned long long value_t);
void operator = (long value_t);
void operator = (unsigned long value_t);
void operator = (float value_t);
void operator = (double value_t);
void operator = (bool value_t);
void operator = (const json_string value_t);
void operator = (const json_char * value_t);
void operator = (const JSONNode & orig);
```

## Assign JSONNode

Assigns JSONNode to a specific value and sets the type according to which version is used:

```
void operator = (char value_t);
void operator = (unsigned char value_t);
void operator = (short value_t);
void operator = (unsigned short value_t);
void operator = (int value_t);
void operator = (unsigned int value_t);
void operator = (long value_t);
void operator = (unsigned long value_t);
void operator = (long long value_t);
void operator = (unsigned long long value_t);
void operator = (float value_t);
void operator = (double value_t);
```
        Constructs a numeric JSONNode.
```
void operator = (bool value_t);
```
        Constructs a boolean JSONNode.
```
void operator = (const json_string value_t);
void operator = (const json_char * value_t);
```
        Constructs a string JSONNode.
```
void operator = (const JSONNode & orig);
```
        Copy assignment, increments the reference counter, will only copy if one of them changes.  To
        make a literal copy, use the duplicate method.

## Option Differences

JSON_REF_COUNT - if this is off, then the copy assignment will fully duplicate the node
JSON_ISO_STRICT - if this option is on then the long long versions are removed

## Parameters

value_t
>        The value of the object
orig
>        The original JSONNode that gets copied

**Complexity**

All assignments run in constant time due to reference counting, if reference counting is not turned on, then the copy assignment is the same as duplicate.

# JSONNode::operator ==

```
bool operator == (char value_t);
bool operator == (unsigned char value_t);
bool operator == (short value_t);
bool operator == (unsigned short value_t);
bool operator == (int value_t);
bool operator == (unsigned int value_t);
bool operator == (long long value_t);
bool operator == (unsigned long long value_t);
bool operator == (long value_t);
bool operator == (unsigned long value_t);
bool operator == (float value_t);
bool operator == (double value_t);
bool operator == (bool value_t);
bool operator == (const json_string value_t);
bool operator == (const json_char * value_t);
bool operator == (const JSONNode & node);
```

**Comparing JSONNode**

Checks if the value held within the nodes are equal.  This ignores things like comments, but for JSON_NODE and JSON_ARRAYs, this is a deep comparison, checking each child too.

**Option Differences**

JSON_ISO_STRICT - if this option is on then the long long versions are removed

**Parameters**

value_t
        The value to compare to
node
        Another node to compare to

**Complexity**

Constant except for JSON_NODEs and JSON_ARRAYs in which case it's linear with respect to size() and size() of each child recursively.

# JSONNode::operator !=

bool operator != (char value_t);
bool operator != (unsigned char value_t);
bool operator != (short value_t);
bool operator != (unsigned short value_t);
bool operator != (int value_t);
bool operator != (unsigned int value_t);
bool operator != (long long value_t);
bool operator != (unsigned long long value_t);
bool operator != (long value_t);
bool operator != (unsigned long value_t);
bool operator != (float value_t);
bool operator != (double value_t);
bool operator != (bool value_t);
bool operator != (const json_string value_t);
bool operator != (const json_char * value_t);
bool operator != (const JSONNode & node);


## Comparing JSONNode

Checks if the value held within the nodes are not equal.  This ignores things like comments, but for JSON_NODE and JSON_ARRAYs, this is a deep comparison, checking each child too.

## Option Differences

JSON_ISO_STRICT - if this option is on then the long long versions are removed

## Parameters

value_t
        The value to compare to
node
        Another node to compare to

## Complexity

Constant except for JSON_NODEs and JSON_ARRAYs in which case it's linear with respect to size() and size() of each child recursively.

## JSONNode::type

```
char type(void);
```

**Return type**

Returns the type of the JSONNode.

**Option Differences**

None

**Parameters**

**Return Value**

The type of the node.

**Complexity**

Constant

## JSONNode::size

size_t size(void);

### Return size

Returns the number of children that the node has.  This should be zero for anything other than JSON_ARRAY or JSON_NODE, but this is only guaranteed with the JSON_SAFE option turned on.  This is because casting may or may not purge the children.

### Option Differences

None

### Parameters

None

### Return Value

The number of children that the node has.

### Complexity

Constant.

## JSONNode::empty
bool empty(void);

**Return empty**

Returns whether or not the node has any children.  If the node is not of JSON_NODE or JSON_ARRAY it will invariably return true.

**Option Differences**

None

**Parameters**

None

**Return Value**

Whether or not the node is empty

**Complexity**

Constant.

## JSONNode::name

json_string name(void);

**Return name**

Returns the name of the node.  If there is no name, then it returns a blank string.

**Option Differences**

None

**Parameters**

None

**Return Value**

The name of the string.

**Complexity**

Constant.

## JSONNode::get_comment

json_string get_comment(void);

**Return comment**

Returns the comment attached to the node

**Option Differences**

JSON_COMMENTS - If this option is turned off, this function is not exposed

**Parameters**

None

**Return Value**

The comment of the node

**Complexity**

Constant.

## JSONNode::as_string
json_string as_string(void);

**Return string value**

Returns the string representation of the node.  This may be different depending on the type of the node

| Function | Description |
|---|---|
| JSON_NULL | "" or "null" depending on how it was created |
| JSON_STRING | The unescaped string value |
| JSON_NUMBER | The number in string form (may be in scientific notation) |
| JSON_BOOL | "true" or "false" |
| JSON_ARRAY | "" |
| JSON_NODE | "" |

Notice that both JSON_NODE and JSON_ARRAY return empty strings.  Use JSONNode::write and JSONNode::write_formatted to output JSON.

**Option Differences**

None

**Parameters**

None

**Return Value**

The string representation of the node.

**Complexity**

Constant.

## JSONNode::as_int

long as_int(void);

**Return int value**

Returns the boolean representation of the node.  This may be different depending on the type of the node

| Function | Description |
|---|---|
| JSON_NULL | 0 |
| JSON_STRING | Undefined |
| JSON_NUMBER | Truncated Value |
| JSON_BOOL | 1 if true, 0 if false |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

If the value is actually a floating point value, then libjson will record an error via it's callback, but will continue on ahead and simply truncate the value.  So 15.9 will be returned as 15.  If JSON_DEBUG is turned on, it will also error if the value of the node is outside the range of the long datatype.

**Option Differences**

None

**Parameters**

None

**Return Value**

The integer representation of the node.

**Complexity**

Constant.

## JSONNode::as_float

json_number as_float(void);

**Return floating point value**

Returns the boolean representation of the node.  This may be different depending on the type of the node

| Function | Description |
|----------|-------------|
| JSON_NULL | 0.0 |
| JSON_STRING | Undefined |
| JSON_NUMBER | Value |
| JSON_BOOL | 1.0 if true, 0.0 if false |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

**Option Differences**

None

**Parameters**

None

**Return Value**

The floating point representation of the node.

**Complexity**

Constant.

# JSONNode::as_bool

bool as_bool(void);

**Return bool value**

Returns the boolean representation of the node.  This may be different depending on the type of the node

| Function | Description |
| --- | --- |
| JSON_NULL | FALSE |
| JSON_STRING | Undefined |
| JSON_NUMBER | Value == 0.0 |
| JSON_BOOL | Value |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

**Option Differences**

None

**Parameters**

None

**Return Value**

The boolean representation of the node.

**Complexity**

Constant.

## JSONNode::as_node

JSONNode as_node(void);

**Return node value**

Returns the node representation of the node.  For anything other than node and array, it simply returns an empty node.  If the caller is an array, it will convert it to a node.

| Function | Description |
|---|---|
| JSON_NULL | Empty node |
| JSON_STRING | Empty node |
| JSON_NUMBER | Empty node |
| JSON_BOOL | Empty node |
| JSON_ARRAY | Array converted to a node |
| JSON_NODE | A copy of the node |

**Option Differences**

None

**Parameters**

None

**Return Value**

The node representation of the node.

**Complexity**

Constant for all except JSON_ARRAYs and JSON_NODEs which are the same as the copy constructor.

## JSONNode::as_array
JSONNode as_array(void);

**Return array value**

Returns the array representation of the node.  For anything other than node and array, it simply returns an empty array.  If the caller is an node, it will convert it to an array by stripping all of the names of each child.

| Function | Description |
|---|---|
| JSON_NULL | Empty node |
| JSON_STRING | Empty node |
| JSON_NUMBER | Empty node |
| JSON_BOOL | Empty node |
| JSON_ARRAY | A copy of the array |
| JSON_NODE | An array of the children |

**Option Differences**

None

**Parameters**

None

**Return Value**

The node representation of the node.

**Complexity**

Constant for anything other than a node.  For a JSON_NODE, it is linear with respect to JSONNode::size, and JSON_ARRAY which is the same as the copy constructor

## JSONNode::as_binary
std::string as_binary(void);

**Return binary value**

Returns the binary value that was part of this node.  It returns it as a std::string, you can use the data() function to retrieve it in binary form.  This allows you to use size() to know how large the binary data is.

| Function | Description |
|----------|-------------|
| JSON_NULL | Undefined |
| JSON_STRING | The binary data from the decoded Base64 |
| JSON_NUMBER | Undefined |
| JSON_BOOL | Undefined |
| JSON_ARRAY | Undefined |
| JSON_NODE | Undefined |

**Option Differences**

JSON_BINARY - If this option is not on, then this function is not accessible.

**Parameters**

None

**Return Value**

The Base64 decoded binary data

**Complexity**

Linear with respect to the length of the binary data

## JSONNode::dump
JSONNode dump(void);

**Dump JSONNode's internal structure**

Returns a JSONNode that exposes all of the inner members of JSONNode.  This is useful for debugging purposes of libjson, but not much else.  It is mostly used to maintain and upgrade libjson.

**Option Differences**

JSON_DEBUG - If this option is not on, then this function is not accessible.

**Parameters**

None

**Return Value**

A JSONNode representation of the inner workers of the JSONNode, showing all of the members and their current state.

**Complexity**

Linear with respect to the size of the tree underneath the node.

## JSONNode::write

json_string write(void);

**Write JSON**

Returns JSON text, with no white space or comments.  Designed to create JSON that is very small, and therefore, faster to send between servers or write to a disk.  The flipside is that it's nearly impossible to read by human eyes.

Only root nodes (JSON_NODE and JSON_ARRAYs) are meant to be written, all others will return a blank string.

**Option Differences**

JSON_WRITER - If this option is not on, then this function is not accessible.

**Parameters**

None

**Return Value**

JSON text of the node being written

**Complexity**

Linear with respect to the size of the tree underneath the node.

## JSONNode::write_formatted

json_string write_formatted(void);

**Write JSON**

Returns JSON text that has been indented and prettied up so that it can be easily read and modified by humans.

Only root nodes (JSON_NODE and JSON_ARRAYs) are meant to be written, all others will return a blank string.

**Option Differences**

JSON_WRITER - If this option is not on, then this function is not accessible.
JSON_COMMENTS - If this option is not turned on, the no comments are written
JSON_WRITE_BASH_COMMENTS - This option will make libjson write only bash style # comments
JSON_WRITE_SINGLE_LINE_COMMENTS - This option will make libjson not write C-style /* */ comments

**Parameters**

None

**Return Value**

JSON text of the node being written

**Complexity**

Linear with respect to the size of the tree underneath the node.

## JSONNode::set_name

void set_name(json_string name_t);

**Setting the node's name**

Sets the name of the JSONNode.

**Option Differences**

None

**Parameters**

name_t
        The name of the node

**Return Value**

None

**Complexity**

Constant

## JSONNode::set_comment

```
void set_comment(json_string comment_t);
```

**Setting the node's comment**

Sets the comment that will be associated with the JSONNode.

**Option Differences**

JSON_COMMENTS - this option is required to expose this function

**Parameters**

comment_t
> The comment attached to the node

**Return Value**

None

**Complexity**

Constant

## JSONNode::clear

void clear(void);

**Clearing a node**

Clears all children from the node.

**Option Differences**

None

**Parameters**

None

**Return Value**

None

**Complexity**

Linear with respect to the size of the tree underneath it

## JSONNode::swap

void swap(JSONNode & other);

**Swaps the contents of nodes**

Swaps the contents of two nodes.  This is very fast because JSONNode is just a wrapper around an internal structure, so it simply swaps pointers to those structures.

**Option Differences**

None

**Parameters**

other
        The node to swap values with

**Return Value**

None

**Complexity**

Constant

## JSONNode::merge

void merge(JSONNode & other);
void merge(unsigned int num, **. . .**);

**Merge the contents of nodes**

It's possible that you may end up with multiple copies of the same node, through duplication and such. To save space, you might want to merge the internal reference counted structure. Obviously, this only makes a difference if reference counting is turned on.

libjson will try and do this efficiently, merging with the internal structure that has the highest reference count.

**Option Differences**

JSON_REF_COUNT - this option is required for this to do anything

**Parameters**

other
      One of the other nodes to merge with.
num
      The number of JSONNodes to merge together
**. . .**
      A list of JSONNode *

**Return Value**

None

**Complexity**

Constant or linear with respect to num.

## JSONNode::preparse
void preparse(void);

**Preparse the json**

libjson's lazy parsing makes parsing JSON that is not entirely used very fast, but sometimes you want to parse it all at once, making the next reads a little faster

**Option Differences**

JSON_PREPARSE - this option does this automatically, so it hides this function

**Parameters**

None

**Return Value**

None

**Complexity**

Linear with respect to the total number of nodes in the JSON

## JSONNode::set_binary

void set_binary(const unsigned char * bin, size_t bytes);

### Set binary data

libjson's built in Base64 encoder will create a JSON_NODE with Base64 encoded binary data, which allows you to send images and files around.  libjson's Base64 encoder is faster than most others, so it's recommended that you use this method rather than other libraries.

### Option Differences

JSON_BINARY - this option is required to use this method and the Base64 method

### Parameters

bin
>        binary data

bytes
>        the number of bytes that the binary data is

### Return Value

None

### Complexity

Linear with respect to the number of bytes

## JSONNode::cast

void cast(char type);

**Cast to a different type**

Will change the node to a different type and do any conversions necessary.

**Option Differences**

None

**Parameters**

type
       New type of node

**Return Value**

None

**Complexity**

Constant except for casting a node to array which is linear with respect to size().

## JSONNode::at

JSONNode & at(json_index_t pos);
JSONNode & at(const json_string & name);
const JSONNode & at(json_index_t pos) const;
const JSONNode & at(const json_string & name) const;

**Getting a child**

This will give you a reference to a child node either at a specific location or by it's name.  This differs from the [] operator because at will throw an std::out_of_bounds exception.

**Option Differences**

None

**Parameters**

pos
> The index of the child node

name
> The name of the child node

**Return Value**

The child at the desired location.

**Complexity**

Constant for the indexed variety, linear for the named variety.

## JSONNode::at_nocase

JSONNode & at_nocase(const json_string & name);
const JSONNode & at_nocase(const json_string & name) const;

**Getting a child case insensitive**

This will give you a reference to a child node by it's name in a case-insensitive way.

**Option Differences**

JSON_CASE_INSENSITIVE_FUNCTIONS is required to use this function

**Parameters**

name
        The name of the child node

**Return Value**

The child at the desired location.

**Complexity**

Linear with respect to size().

## JSONNode::operator []

JSONNode & operator [] (json_index_t pos);
const JSONNode & operator [] (json_index_t pos) const;
JSONNode & operator [] (const json_string & name);
const JSONNode & operator [] (const json_string & name) const;

### Getting a child

This will give you a reference to a child node either at a specific location or by it's name.  Asking for a node that is not there will result in undefined behavior.

### Option Differences

None

### Parameters

pos
>     The index of the child node

name
>     The name of the child node

### Return Value

The child at the desired location.

### Complexity

Constant for the indexed variety, linear for the named variety.

# JSONNode::reserve

void reserve(json_index_t size);

**Reserving space**

This function reserves children space, this makes the program faster and use less memory as it doesn't have to keep allocating new memory when it runs out.

**Option Differences**

None

**Parameters**

size
        The size to reserve

**Return Value**

None

**Complexity**

If there is already some children it's linear with respect to size() unless the reserved amount is less than the current capacity, in which case it's constant.  If there are no children, then it's constant.

## JSONNode::push_back

void push_back(const JSONNode & node);

**Adding a child**

This function pushes a new child node on the back of the child list.  This method copies the child, so altering the parameter later will not affect the one in the children.

**Option Differences**

None

**Parameters**

node
> The node to be added as a child

**Return Value**

None

**Complexity**

Depends on if new memory is needed to be allocated.  This function would have the same complexity as push_back for a vector plus the copy constructor.  If JSON_LESS_MEMORY is defined, then this operation is always linear with respect to size(), unless reserve had been used, in which case it's constant while smaller than that reserved amount .  Otherwise it is linear if the node is currently full, otherwise it's constant.

## JSONNode::pop_back

JSONNode pop_back(json_index_t pos);
JSONNode pop_back(const json_string & name);

**Getting a child**

This will give remove a JSONNode from it's parent and return it to you.

**Option Differences**

None

**Parameters**

pos
> The index of the child node

name
> The name of the child node

**Return Value**

The child at the desired location by value.

**Complexity**

Linear based on size() for the variety using the name, but linear based on size() minus pos.  This is because the references of the other children have to be shifted.

## JSONNode::pop_back_nocase

JSONNode pop_back_nocase(const json_string & name);

**Getting a child**

This will give remove a JSONNode from it's parent and return it to you.

**Option Differences**

JSON_CASE_INSENSITIVE_FUNCTIONS is required to use this function

**Parameters**

name
    The name of the child node

**Return Value**

The child at the desired location by value.

**Complexity**

Linear based on size().

## JSONNode::find

iterator find(const json_string & name);

**Getting a child**

Searches through the children and finds an iterator to it.  This function returns JSONNode::end() if the child does not exist.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

name
       The name of the child node

**Return Value**

An iterator that points to the child requested, or end() if it wasn't found.

**Complexity**

Linear based on size().

## JSONNode::find_nocase

iterator find_nocase(const json_string & name);

**Getting a child**

Searches through the children and finds an iterator to it.  This function returns JSONNode::end() if the child does not exist.

**Option Differences**

JSON_ITERATORS and JSON_CASE_INSENSITIVE_FUNCTIONS must be on to use this function

**Parameters**

name
        The name of the child node

**Return Value**

An iterator that points to the child requested, or end() if it wasn't found.

**Complexity**

Linear based on size().

## JSONNode::erase

iterator erase(iterator pos);
iterator erase(iterator start, const iterator & end);
reverse_iterator erase(reverse_iterator pos);
reverse_iterator erase(reverse_iterator start, const reverse_iterator & end);

**Removing child(ren)**

iterator erase(iterator pos);
reverse_iterator erase(reverse_iterator pos);
　　　　Erases a single child and returns the iterator to the next one.
iterator erase(iterator start, const iterator & end);
reverse_iterator erase(reverse_iterator start, const reverse_iterator & end);
　　　　Erases a set of children, allowing for quick dumps of unneeded children.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

pos
　　　　The position of the node that is to be deleted
start
　　　　The position of starting node to be deleted (inclusively)
end
　　　　The position of ending node to be deleted (inclusively)

**Return Value**

An iterator that points to the child after the deleted one, or end()/rend()

**Complexity**

Linear based on size() minus where it starts from due to memory shifting.

## JSONNode::insert

iterator insert(iterator pos, const JSONNode & node);
iterator insert(iterator pos, const iterator & start, const iterator & end);
iterator insert(iterator pos, const reverse_iterator & start, const reverse_iterator & end);
iterator insert(iterator pos, const const_iterator & start, const const_iterator & end);
iterator insert(iterator pos, const const_reverse_iterator & start, const const_reverse_iterator & end);
reverse_iterator insert(reverse_iterator pos, const JSONNode & node);
reverse_iterator insert(reverse_iterator pos, const reverse_iterator & start, const reverse_iterator & end);
reverse_iterator insert(iterator pos, const iterator & start, const iterator & end);
reverse_iterator insert(reverse_iterator pos, const const_iterator & start, const const_iterator & end);
reverse_iterator insert(reverse_iterator pos, const const_reverse_iterator & start, const const_reverse_iterator & end);

### Adding child(ren)

iterator insert(iterator pos, const JSONNode & node);
 reverse_iterator insert(reverse_iterator pos, const JSONNode & node);
> These functions place a new child into your node before the node pointed to by the iterator.

iterator insert(iterator pos, const iterator & start, const iterator & end);
iterator insert(iterator pos, const reverse_iterator & start, const reverse_iterator & end);
iterator insert(iterator pos, const const_iterator & start, const const_iterator & end);
iterator insert(iterator pos, const const_reverse_iterator & start, const const_reverse_iterator & end);
reverse_iterator insert(reverse_iterator pos, const reverse_iterator & start, const reverse_iterator & end);
reverse_iterator insert(iterator pos, const iterator & start, const iterator & end);
reverse_iterator insert(reverse_iterator pos, const const_iterator & start, const const_iterator & end);
reverse_iterator insert(reverse_iterator pos, const const_reverse_iterator & start, const const_reverse_iterator & end);

> These functions will copy a set of children from one node to another.  These all start inserting at start and increment it until ti's end, so if you are reverse iterating, you can insert them in reverse order.

All of these functions copy the nodes, they remain where they are as well.

### Option Differences

JSON_ITERATORS must be on to use this function

### Parameters

pos
> The position to insert before

node
> The node to be inserted

start
> Position to start copying (inclusively)

end
> Position to stop copying (inclusively)

### Return Value

An iterator that points to the first inserted node

**Complexity**

Linear based on size() minus where it starts from due to memory shifting plus end minus start for the ranged variety.

## JSONNode::begin

iterator begin(void);
const_iterator begin(void) const;

**Getting start of children**

This function gets you an iterator pointing to the beginning of the children.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

None

**Return Value**

An iterator that points to the first child

**Complexity**

Constant

## JSONNode::end

iterator end(void);
const_iterator end(void) const;

**Getting end of children**

This function gets you an iterator pointing past the end of the children.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

None

**Return Value**

An iterator that points past the last child

**Complexity**

Constant

## JSONNode::rbegin

reverse_iterator rbegin(void);
reverse_const_iterator rbegin(void) const;

**Getting start of children**

This function gets you an iterator pointing to the beginning of the children in reverse order.

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

None

**Return Value**

An iterator that points to the first child in reverse order (actually the last child.)

**Complexity**

Constant

## JSONNode::rend

reverse_iterator rend(void);
reverse_const_iterator rend(void) const;

**Getting start of children**

This function gets you an iterator pointing past the end of the children in reverse order

**Option Differences**

JSON_ITERATORS must be on to use this function

**Parameters**

None

**Return Value**

An iterator that points past the last child in reverse order

**Complexity**

Constant

## JSONNode::set_mutex

void set_mutex(void * mutex);

**Set a lock**

      This function attaches a mutual exclusion lock for the node.  It can be locked and unlocked by using the lock and unlock functions,  The reason for this is that because libjson's complex reference counting and internal magic, it is difficult to be sure that you track data correctly, so libjson will pass this mutex around it's internal system and track it for you.
      You can pass NULL to this function to unset a mutex.
      This method will filter the mutex down to all children as well, and anything that you add to it later.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this function
JSON_MUTEX_MANAGER - if this option is on, then libjson will automatically delete this mutex when it
      runs out of references to it

**Parameters**

mutex
      A pointer to a mutex of any type (POSIX, Windows…)

**Return Value**

None

**Complexity**

Linear with size() and size() of each child recursively.

# JSONNode::lock
void lock(int threadID);

**Lock a mutex**

This function locks the mutex attached to a node, and locks anything sharing the lock in a different thread.  The reason for the threadID parameter is that because libjson passes around structures behind the scenes it might be possible to try and lock the same lock in multiple places in the same critical section.  Counting locks per thread eliminates this problem.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this function

**Parameters**

threadID
         A unique id for this thread

**Return Value**

None

**Complexity**

Constant

## JSONNode::unlock
void unlock(int threadID);

**Unlock a mutex**

This function unlocks the mutex attached to a node.  If the mutex has been locked multiple times by the same thread, then it will wait until all locks have been released before actually doing the unlock.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this function

**Parameters**

threadID
A unique id for this thread

**Return Value**

None

**Complexity**

Constant

**libjson**

libjson is a namespace that holds a few functions that are part of libjson, but separate from JSONNode.  These methods include json text functions and callback registration.

**JSON Functions**

| Function | Description |
|---|---|
| parse | Parses json |
| strip_white_space | Removes all white space and comment |
| validate | Validates json |

**Callback Registration**

| Function | Description |
|---|---|
| register_debug_callback | Registers error callback |
| register_mutex_callbacks | Register mutex callbacks |
| set_global_mutex | Set the global mutex |
| register_memory_callbacks | Registers the memory callbacks |

**Callback Types**

| Function | Description |
|---|---|
| json_error_callback_t | typedef void (* json_error_callback_t)(const json_string &) |
| json_mutex_callback_t | typedef void (* json_mutex_callback_t)(void *) |
| json_malloc_t | typedef void * (* json_malloc_t)(size_t) |
| json_realloc_t | typedef void * (* json_realloc_t)(void *, size_t) |
| json_free_t | typedef void (* json_free_t)(void *) |

## libjson::parse

JSONNode parse(const json_string & json);

**Parse JSON text**

      This function parses JSON text and returns you a JSONNode which is the root node of the text that you just passed it.  If bad JSON is sent to this method it will throw a std::invalid_argument exception.

**Option Differences**

JSON_SAFE - a node of type JSON_NULL will be returned if the json was invalid

**Parameters**

json
      JSON text

**Return Value**

The root node of the text

**Complexity**

Linear depending on json.length()

## libjson::strip_white_space

json_string strip_white_space(const json_string & json);

**Parse JSON text**

      This function removes anything that the JSON standard defines as white space, including extra tabs, spaces, formatting, and comments.  This makes this function useful for compressing json that needs to be stored or sent over a network.

**Option Differences**

None

**Parameters**

json
      JSON text

**Return Value**

Valid JSON that is free of all white space

**Complexity**

Linear depending on json.length()

## libjson::validate

JSONNode validate(const json_string & json);

**Validate JSON text**

       This function validates the text by parsing it completely and looking for anything that is malformed.  If bad JSON is sent to this method it will throw a std::invalid_argument exception, otherwise it returns the root node of the text.

**Option Differences**

JSON_VALIDATE and JSON_SAFE must be on to use this feature

**Parameters**

json
       JSON text

**Return Value**

The root node of the text

**Complexity**

Linear depending on json.length() and the number of nodes within the text.

## libjson::register_debug_callback

void register_debug_callback(json_error_callback_t callback);

### Register error callback

This callback allows libjson to tell you exactly what is going wrong in your software, making debugging and fixing the problem much easier.

### Option Differences

JSON_DEBUG must be on to use this feature

### Parameters

callback
A method that must have this prototype: static void callback (const json_string & message)

### Return Value

None

### Complexity

Constant

## libjson::register_mutex_callbacks

void register_mutex_callbacks(json_mutex_callback_t lock, json_mutex_callback_t unlock, void * manager_mutex);
void register_mutex_callbacks(json_mutex_callback_t lock, json_mutex_callback_t unlock, json_mutex_callback_t destroy, void * manager_mutex);

**Register mutex callbacks**

This callback allows libjson to lock mutexes that you assign to JSONNode structures.  Because libjson has no idea what kind of mutex you gave it, it can't lock it without the help of a callback.  This method is required to run before any locking can take place.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this feature
JSON_MUTEX_MANAGER must be on to use the second variant

**Parameters**

lock
>A method that locks a mutex and must have this prototype: static void callback (void * mutex).

unlock
>A method that unlocks a mutex and must have this prototype: static void callback (void * mutex).

destroy
>A method that deletes a mutex and must have this prototype: static void callback (void * mutex). This parameter is only allowed if JSON_MUTEX_MANAGER is defined, otherwise, you are responsible for cleaning up you mutexes.

manager_mutex
>A mutex that libjson will need to use to handle all of the mutexes, as even without JSON_MUTEX_MANAGER, some management takes place and libjson has a few critical sections when threading.  May not be the same as the global mutex.

**Return Value**

None

**Complexity**

Constant

## libjson::set_global_mutex
void set_global_mutex(void * mutex);

**Register library-wide mutex**

This function sets a global mutex for libjson.  This is not the same as the manager mutex that you are required to register in register_mutex_callbacks.  This mutex is a fallback mutex that gets locked when you attempt to lock a node that has no mutex attached to it.  You may wish to omit assigning mutexes for each node and just let them all use the global one.

**Option Differences**

JSON_MUTEX_CALLBACKS must be on to use this feature

**Parameters**

mutex
A mutex that libjson will use when it doesn't have any to use

**Return Value**

None

**Complexity**

Constant

## libjson::register_memory_callbacks

void register_memory_callbacks(json_malloc_t allocer, json_realloc_t reallocer, json_free_t freer);

**Register memory handling**

This function sets callbacks that libjson will use to allocate memory. This allows you to keep a tight control of memory allocation, or use things like memory pool and such. These methods are currently not allowed to return null or less than it asked for, as it will cause instability in libjson.

**Option Differences**

JSON_MEMORY_CALLBACKS must be on to use this feature

**Parameters**

allocer
> A method that must have this prototype: static void * alloc(size_t bytes)

reallocer
> A method that must have this prototype: static void * realloc(void * buffer, size_t bytes)

freer
> A method that must have this prototype: static void dealloc(void * buffer)

**Return Value**

None

**Complexity**

Constant

# JSON_TEXT

## Converts text

This is a marco that you can use to wrap around text that you pass into libjson.  It automatically prepends an L in unicode mode and does nothing when not in unicode mode.  This allows you to use the same code for unicode and non unicode software.

## Optimizing libjson

Once your program is written and functioning with libjson, it is time to optimize it. libjson's options are the best way to do this, however if you are using a library, then you can also edit the make file and change the compiler options. Obviously you should never try to optimize with a debug build, it's doing a lot of extra checks that are unneeded in release.

## Compiler Options

You can see the default compile options in the building libjson section of this document. -O3 isn't always optimal, because it's such a small library, caching may actually make -Os the fastest, it depends on how much cache you have, what kind of computer, and what libjson options are turned on.

There are also many optimizations that you can do that are compiler specific. libjson does a decent amount of string functions internally, so turning on reference counted or copy on write strings may result in a performance increase, both faster due to less copying and less memory.

## C++ Optimizations

If using C++, you can embed libjson and compile it into your program. This saves any time at the interface. The C interface is fairly rigid, and everything has to pass through the library interface, creating a choke-point. But in C++, many of the functions get inlined, so that little extra overhead is gone. The compiler is also free to do things like return by reference and other little tricks that you can't get with the json_ interface.

## JSONOptions.h Optimizations

You can also optimize libjson by changing the options. Here are some common optimizations.

### Speed

If you know that you will always be receiving proper json, or json written by your libjson engine (like if you are using it to store and read a configuration file), then JSON_SAFE is probably not needed. That option does a lot of checks to make sure that the json isn't corrupt and guards if it is. If there is no need for these guards, don't bother with them.

The JSON_MEMORY_CALLBACKS option can help you dramatically speed up your program. libjson allocates a lot of memory using malloc, but those calls can be rerouted to callbacks, allowing you to utilize your own memory management system. A memory pool can speed up a program.

### Memory

If JSON_STDERROR is on, libjson prints out any errors to std::cerr, which requires iostream. iostream adds considerable size to the library. Removing the JSON_STDERROR option will compile the library without iostream in it.

If you are parsing huge amounts of JSON, or are working with very limited memory and need libjson to use less of it, turn on the JSON_LESS_MEMORY option. This will cause libjson to be far more stringent with it's memory and clean it up faster. It also makes it slightly slower though because it allocates and deallocates memory more often.

Manage you own memory. Turn JSON_MEMORY_MANAGE off if you turned it on and delete your memory when you are done using it instead of just letting it sit around. libjson utilizes a map to keep

track of memory, this can grow quiet a bit if not flushed often enough.  Managed memory is in libjson for ease of use, not performance.

If you can bring down the size of the indexes, it may save some bytes.  By turning on and changing the JSON_INDEX_TYPE option, you can choose a different type of all of the indexing.  This cuts down on the size of the internal structure, but be careful, you have to give enough space to hold all of your nodes.  If you are sure that no node will ever hold more than 255 subitems, you can even use an unsigned char.

You might be able to further decrease the memory required and the size of the library if you use your own string class.  JSON_STRING_HEADER lets you drop in your own string class which you can heavily optimize for your needs.

**Overall**

If your program almost always uses most or all of the json that it receives, then doing parsing as needed might not make sense, and you might want all parsing done at once.  JSON_PREPARSE will force libjson to do this.  This makes the initial parse command a little slower, but every subsequent request is faster.  It also drops the amount of memory libjson uses slightly.

If you turned JSON_REF_COUNT off, and can find a way to turn it back on, do it.  This cuts down on both the amount of memory used when copying objects, but also saves copy time, which sometimes can be quite expensive.  The common reason to turn this option off is for thread safety.  It is recommended that you use the mutex callback system instead.

Write your own string class, or find a better one.  There are many string classes out there that perform better than STL in some cases and are drop in and forget.  Define JSON_STRING_HEADER to point to a header where you typedef your relpacement string as json_string.  libjson has no control over how the strings work, so you can often specialize a string type for this case.  The majority of both time and memory used by libjson is spent doing string manipulation.

# Changelog

6.2.1 (11/19/10)
>   Fix a few documentation slips
>   Fixed bug in All Test Suite options
>   Defined as_int and as_float for strings
>   reserve now is a noop when JSON_LESS_MEMORY is on
>   Added JSON_BOOL_TYPE option for the C interface

6.2.0 (11/18/10)
>   Fix problem with escaping names sometimes
>   Added JSON_ESCAPE_WRITES option to make escaping special chars optional
>   Added JSON_STRING_HEADER option to drop in custom string classes
>   Added JSON_NO_C_CONSTS option
>   Memory enhancements throughout
>   Added Salvor Hardin's fix for missing snprintf in VC++
>   Added C++ example in the Getting Started folder

6.1.2 (11/11/10)
>   Fixed bug where unicode wasn't writing escaped UTF correctly
>   Added JSON_TEXT macro to the documentation

6.1.1 (11/10/10)
>   Fixed bug with consecutive surrogate pairs not being read correctly
>   Changed license to FreeBSD

6.1.0 (10/16/10)
>   Made buffers for converting numbers to strings solved at compile time
>   Minor performance tweaks throughout
>   Better examples thanks to Chris Larsen
>   Fixed a few mistakes in documentation

6.0.0 (9/11/10)
>   Complete rewrite of library
>   More STL-like C++ interface
>   More standard C interface
>   Must faster
>   Less Memory
>   Must more customizable
>   Huge Test Suite
>   Complete cplusplus.com-like documentation

5.2.4 (8/24/10)
>   Added memory cleanup in example.cpp

5.2.3 (8/24/10)
>   Fixed safety catch in mismatched quotes

5.2.2 (8/20/10)
>   Added safety catch for mismatched quotes

5.2.1 (8/6/10)
>   Fixed bug 3040432 (not mallocing enough memory in white space stripper)

5.2 (7/7/10)

Changed some size_t to unsigned ints for uniformity

Made some functions faster

Fixed problem with key/value pairs being accepted in arrays

Fixed problem where DEBUG mode would give false Children errors

Fixed compiling error under Linux

5.1 (7/6/10)

Fixed bug where duplication arrays or nodes wouldn't work

Added JSON_LESS_MEMORY compiling option

Replaced std::vector with custom container

5.0 (6/27/10)

Added a JSON_SAFE compiling option

Added JSON_PREPARSE compiling option

Fixed bug with name_encoded flag not dumping correctly

Nodes now use 5% - 20% less memory

4.5 (6/18/10)

Made unicode slightly more efficient

Fixed some documentation

Added GetMemoryUsage method to debugger

Nodes now use 8% less memory

4.4 (6/17/10)

Fixed bug where false values were not always flagged as Fetched

Fixed typecast when converting double to int

Used a union for boolean and double values for smaller memory footprint

4.3 (6/16/10)

new makefiles for each platform

Removed unused declarations

4.2 (4/7/10)

Fixed makefile

Removed unused declarations

4.1 (4/5/10)

Full support for surrogate pairs

makefile for linux

Better documentation

4.0 (4/21/10)

Numbers not get fetched lazily too

Added full unicode support by adding a JSON_UNICODE compile option

Added unicode targets in Code::Blocks projects

Prebuilt libraries / suites now not included to make downloads faster

3.0 (4/20/10)

Fixed bug with Root Arrays printing an extra comma

Root arrays now also lazily fetched

Unified array parsing routine

Decent speed and memory optimizations

NodeAsFloat now returns a double, not a single

Replaced profiler with Dump method

Dump now uses allocated string space instead of length
Added Preparse function
Made header safeties more obscure to not interfere with other headers
2.2 (4/18/10)
More technical documentation
More speed in initial parsing
Added profiler and raw memory dumps to Debug version
2.1 (4/15/10)
Added support for escaping object names
Added support for bash-style comments
Made writing strings without escaping faster
Made initial parse faster and uses less memory
2.0 (4/7/10)
Added support for Octal string escaping
Added support for both single and multiline C-style comments
1.0 (4/1/10)
Initial release
Renamed LiveJSON to libjson and added Test Suites

# License