

Advanced C Programming

Summer 2022 ECE 264 :: Purdue University

[Home](#)
[Schedule](#)
[Syllabus](#)
[Resources](#)
[Standards](#)
[Scores](#)


⚠ This is a PAST SEMESTER (Summer 2022).

JSON 5: parse objects

Due 8/5

Learning goals

You will learn or practice how to:

1. Work with binary search trees as part of a larger program.
2. Refactor existing code, avoiding breaking previous tests.
3. Submit code that is has finished some functionality.

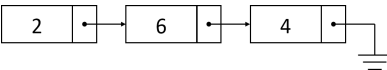
Overview

In this homework, you will adapt your code from [HW15](#) to allow parsing JSON objects. At this point, you will have the majority of the functionality of a JSON parser working. Refer to the previous parts for specifications on valid and invalid JSON.

This assignment serves two purposes:

1. Adds the major missing functionality to your JSON parser. This project might be good for a resume.
2. For those who did not finish previous JSON homeworks, this is one last chance to submit JSON for credit.

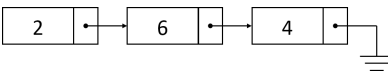
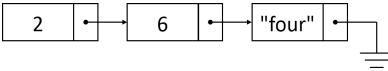
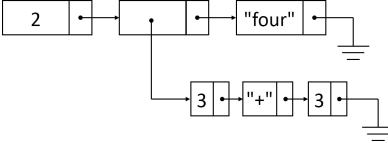
A JSON file contains text that represents a number, a string, a list, an object, a boolean, or null. Here are some examples:

type	json	data
number	10	10
string	"ten"	"ten"
list	[2, 6, 4]	
list	{"key1": "value", "key2": true}	Imagine a lovely picture of a binary search tree here. There is an image from lecture on 8/2, you can find it on the schedule page.
boolean	true	true
null	null	

[Learning goals](#)
[Overview](#)
[Linked lists](#)
[Getting Started on EC04](#)
[Requirements](#)
[Submit](#)
[Pre-tester ●](#)
[Q&A](#)
[How can I structure my tests?](#)
[That string looks super messy, is there a nicer way we can write it?](#)
[I finished HW15, can I submit this homework without doing objects?](#)
[Isn't this requirement unfair for those who finished HW15?](#)
[Can I turn in this assignment late?](#)
[Updates](#)

Linked lists

As in C, whitespace is ignored (except within a string). Lists and objects can both contain any type of data, thus all the following are valid:

type	json	data
list	[2, 6, 4]	
list	[2, 6, "four"]	
list	[2, [3, "+", 3], "four"]	

Getting Started on EC04

Okay to hand-copy/adapt the test code in this section, if you understand it.

We are providing test code for some steps of this process. You are free to test your code however you like. The test code provided here is optional. Use it if you find it helpful. You are responsible for the correctness and comprehensiveness of the tests that you submit. Since these were written as part of this web page, errors are possible. Let us know if you find any.

Tip: If you are working on ecegrid, you can use some snippets to save you some typing. From insert mode, type the snippet abbreviation (below) and press Tab.

- `mu` → new test function
- `mur` → `mu_run(`
- `muc` → `mu_check(`

1. The starter code for EC04 is the same as previous JSON assignment. You will be required to modify the header to add the new functions.
2. Start by creating a directory and cp to copy your code from [HW15](#).
e 264get EC04 to fetch the starter code.

```
you@ecegrid-thin1 ~/
$ cd 264

you@ecegrid-thin1 ~/264/
$ mkdir ec04

you@ecegrid-thin1 ~/264/
$ cp hw15/* -v -t ec04/

you@ecegrid-thin1 ~/264/
$ cd ec04


you@ecegrid-thin1 ~/264/ec04
$
```

3. Start by fixing functionality from previous JSON parts that you did not finish in time for [HW15](#). The majority of the credit for ec04 will be for previous functionality.
4. Make a submission. You will get credit for ec04 as long as you have more functionality finished than on [HW15](#).
5. Let me reiterate, finish previous functionality first before you worry about objects. You do not have to be finished to make a submission, as long as it compiles and passes some of your tests and it
6. Start by creating a helper to insert key and an element into a binary search tree. Recommended signature: `void _bst_insert(BSTNode** a_root, char const* key, Element element)`.
7. We recommend finishing `get_element(...)` next. You can test it by creating binary trees manually using your helper function.
8. Test and submit.
9. Next, update `print_element(...)` to support objects. This will help you debug.
10. Test and submit.
11. Create `parse_object(...)`
 - a. Start by implementing code to parse an empty object.
 - b. Test and submit.
 - c. Next, create code to parse an element with a single key and value.
 - d. Test and submit.
 - e. Next, update the code to parse two key and value pairs.
 - f. Test and submit.
 - g. Update the code to parse any number of key and value pairs.
 - h. Test and submit.
 - i. Finally, add code to handle any invalid cases you did not handle before.
 - j. Test and submit.
12. Update `parse_element(...)` to support JSON objects.
13. Create recursive helper function to free the binary search tree. Recommended signature: `void _free_tree(BSTNode** a_root)`.
14. Finally, update `free_element(...)` to support JSON objects.
15. Test that there are no memory leaks, even for incorrect input.

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents
json.h	type definitions Element <ul style="list-style-type: none"> Everything that was in <i>element</i> before. To the type enum, add one new type: <code>ELEMENT_OBJECT</code>. To the struct, add one new field: <code>BSTNode* as_object</code>. BSTNode struct type with 4 fields: <code>key (char*)</code> , <code>element (Element)</code> , <code>left</code> , and <code>right (BSTNode*)</code> . Sort the tree using <code>key</code> , no need to consider <code>element</code> when sorting.
	function declarations One for each required function in your <code>json.c</code> <ul style="list-style-type: none"> Do not include helpers (if any) here.
	json.c functions All functions that were required in <code>json.c</code> previously. Only new functions or functions that were changed will be listed below.
	<code>get_element(BSTNode* root, char const* key)</code> → return type: <code>Element*</code> Finds the node in the binary search tree which matches the key. <ol style="list-style-type: none"> If found, return the address of the element in the node. If not found, return <code>NULL</code>. Hint: you can write this either using recursion or a loop. Hint: this function will be helpful in testing <code>parse_object(...)</code>. Note that this function originally had a typo in the header with the first parameter being <code>BSTNode**</code> instead of <code>BSTNode*</code>. You are free to implement the function either way without losing credit, however a parameter of type <code>BSTNode*</code> will be easier.

file	contents
	<p>parse_object(BSTNode** a_root, char const** a_pos) → return type: bool Set *a_head to the head of a linked list of Element objects.</p> <ol style="list-style-type: none"> 1. Caller is responsible for freeing the memory if parse_object(...) returns true. 2. Linked list consists of a '{', followed by 0 or more key-value pairs separated by commas, and finally a '}'. Each key-value pair consists of a string key, a character, then a JSON-encoded element (integer, string, list, object, boolean, or null). See the examples above. (There will be no HBB on this definition, unless there is something truly wrong and/or grossly unclear.) There may be any number/amount of whitespace characters (' ', '\n', or '\t'), before/after any of the element and before/after the key. 3. Return true if a properly formed object was found. *a_pos should be set to the next character in the input string, <i>after</i> the object that was just parsed. <ol style="list-style-type: none"> 1. Ex: parse_object(...) should return true for <code>{}</code>, <code>{"number": 123}</code>, <code>{"string": "Hello", "list": [1, 2, 3]}</code>, <code>{"nested_object": { "integer": 5 } }</code>, and <code>{"string_contains": "trailing_characters" }ABC</code>. 4. Return false if an object was <i>not</i> found (i.e., syntax error in the input string). *a_pos should refer to the unexpected character that indicated a problem. <ol style="list-style-type: none"> 1. Ex: parse_object(...) should return false for <code>A{}</code>, <code>{"key": 1,, "value": 2}</code>, <code>\{\{\,</code>, <code>"key": 1}</code>, and <code>{ "key": 1 "key2": 2}</code>. <code>{ "key" 1 }</code>. 5. The parsed elements are stored in a binary search tree sorted by the key values. 6. If duplicate keys are found, replace the value in the old node with the new element parsed. For example, <code>{"duplicate": 123, "duplicate": 456}</code> should produce the same result as <code>{"duplicate": 456}</code>,  do not forget to free the old element. 7. Whenever parse_object(...) returns false, do not modify *a_root, and free any heap memory that was allocated prior to discovery of the error. 8. Hint: you already have logic written to parse a string, reuse that logic instead of writing it again. <p>parse_element(Element* a_element, char const** a_pos) → return type: bool</p> <ol style="list-style-type: none"> 1. First, eat any whitespace at *a_pos. <ol style="list-style-type: none"> 1. “Eat whitespace” just means to skip over any whitespace characters (i.e., increment *a_pos until isspace(**a_pos)==false). 2. Next, decide what kind of element this is. <ol style="list-style-type: none"> 1. If it's a digit (isdigit(**a_pos)) or hyphen ('-'), set the element's type to ELEMENT_INT and call parse_int(&(a_element -> as_int), a_pos). 2. If it's a string (**a_pos==' '), then set the element's type to ELEMENT_STRING and call parse_string(&(a_element -> as_string), a_pos). 3. If it's a list (**a_pos == '['), then set the element's type to ELEMENT_LIST and call: parse_list(&(a_element -> as_list), a_pos). 4. If it's a boolean (**a_pos == 't' or **a_pos == 'f'), then set the element's type to ELEMENT_BOOL and call: parse_bool(&(a_element -> as_bool), a_pos). 5. If it's null (**a_pos == 'n'), then set the element's type to ELEMENT_NULL, set the element's as_null to NULL, and call: parse_null(a_pos). 6. If it's an object (**a_pos == '{'), then set the element's type to ELEMENT_OBJECT and call: parse_object(&(a_element -> as_object), a_pos). 3. Return whatever was returned by the relevant function. <ul style="list-style-type: none"> ◦ If none of those functions was called—i.e., if the next character was none of the expected characters, then return false. 4. Do not modify *a_pos directly in parse_element(...), <i>except for eating whitespace</i>. <ul style="list-style-type: none"> ◦ *a_pos can—and should—be modified in the relevant parse functions. 5. Caller is responsible for freeing memory by calling free_element(...) whenever parse_element(...) returns true. 6. Whenever parse_element(...) returns false, do not modify *a_element, and free any heap memory that was allocated prior to discovery of the error. <p>print_element_to_file(Element element, FILE* file) → return type: void Given an Element object, print it in JSON notation to the passed file.</p> <ol style="list-style-type: none"> 1. Spacing is up to you, as long as it is valid JSON. 2. If element is an integer, print it using fprintf(...). 3. If element is a string, then print it (with double-quotes) using fprintf(...).

file	contents
	<p>4. If element is a list, print a '['. Then print each element in the list using <code>print_element_to_file(...)</code> (recursively), separated by commas. Finally, print ']'. 5. If element is an object, print a '{'. Then print each key-value pair in the object using <code>fprintf(...)</code>, followed by ':', followed by the element using <code>print_element_to_file(...)</code> (recursively). Separate each key-value pair using commas. Finally, print '}'. 6. If element is a boolean or null, print it using a string constant.</p> <p>free_element(Element element) → <i>return type</i>: void Free the <i>contents</i> of the Element, as needed.</p> <ol style="list-style-type: none"> 1. If it contains a string, free the string. 2. If it contains a linked list, free the list, including all elements. 3. If it contains a binary search tree, free each node in the tree, including all elements. 4. ⚠ Do not attempt to free the Element object itself. <code>free_element(element)</code> only frees dynamic memory that element refers to.
test_json.c	<p>functions</p> <p>main(int argc, char* argv[]) → <i>return type</i>: int Test your all of the above functions using your <code>miniunit.h</code>.</p> <ul style="list-style-type: none"> • This should consist primarily of calls to <code>mu_run(_test_...)</code>. • 100% code coverage is required. • Your <code>main(...)</code> must return <code>EXIT_SUCCESS</code>.

2. You may ignore any trailing characters in the input string in any function starting with `parse_`, as long as the input *starts* with a well-formed JSON element.
 - Acceptable: `123AAA`, `"12"AAA`, `"12"`, `[`,
Do not ignore trailing characters in `read_json(...)`, other than whitespace.
3. You only need to support the specific features of JSON that are explicitly required in this assignment description. You do *not* need to support unicode (e.g., "萬國碼", "يونيكود", "യൂണികോഡ്"), backslash escapes (e.g., "\n"), embedded quotes (e.g., "He said, \"Roar!\""), floating point numbers (e.g., 3.1415), non-decimal notations (e.g., 0xdeadbeef, 0600)
4. Do not add helper functions to `json.h`.
5. There may be no memory faults (e.g., leaks, invalid read/write, etc.), even when `parse_...` (...) or `read_json(...)` return false.
6. The following external header files, functions, and symbols are allowed.

header	functions/symbols	allowed in...
stdbool.h	bool, true, false	json.c, test_json.c
stdio.h	fprintf, fputc, fputs, stdout, fflush, fopen, fclose, fseek, ftell, rewind, fread, fgetc	json.c, test_json.c
assert.h	assert	json.c, test_json.c
ctype.h	isdigit, isspace	json.c, test_json.c
stdlib.h	EXIT_SUCCESS, abs, malloc, free, size_t	json.c, test_json.c
string.h	strncpy, strchr, strlen, strcmp	json.c, test_json.c
limits.h	INT_MIN, INT_MAX	json.c, test_json.c
miniunit.h	anything	test_json.c
clog.h	anything	json.c, test_json.c

For `miniunit.h` and `clog.h`, you can use anything from HW05. You are welcome to change them to your liking, and/or add more in the same spirit. All others are prohibited unless approved by the instructor. Feel free to ask if there is something you would like to use.

7. Submissions must meet the [code quality standards](#) and the [course policies](#) on homework and academic integrity.

Submit

To submit EC04 from within your `ec04` directory, type `264submit EC04 json.c json.h test_json.c miniunit.h clog.h Makefile *.json`

If your code does not depend on `miniunit.h` or `clog.h`, those may be omitted. Your `json.h` will most likely be identical to the starter. Makefile will not be checked, but including it may help in case we need to do any troubleshooting. Make sure to submit all required json files required by your tests. It is your responsibility to ensure the tester has all the files required to test. You are encouraged to modify your Makefile to do so.

In general, to submit any assignment for this course, you will use the following command:

```
264submit ASSIGNMENT FILES...
```

Submit often and early, even well before you are finished. Doing so creates a backup that you can retrieve in case of a problem (e.g., accidentally deleted your files).

To retrieve your most recent submission, type `264get --restore ASSIGNMENT` (e.g., `264get --restore ec04`).

To retrieve an earlier submission, first type `264get --list ASSIGNMENT` to view your past submissions and find the timestamp of the one you want to retrieve. Then, type `264get --restore -t TIMESTAMP ASSIGNMENT`.

Scores will be posted to the [Scores](#) page after the deadline for each assignment.

Pre-tester ●

The pre-tester for EC04 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test ec04
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.

- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤ 24 times in a 24-hour period. (This is not implemented yet but will be added.)

Q&A

1. How can I structure my tests?

Here's a start. (We may add to this at some point.)

```
// OK TO COPY / ADAPT this snippet---but ONLY if you understand it completely.
// ⚠ Do not copy blindly.
//
// This test is nowhere near adequate on its own. It is provided to illustrate how to
// use helper functions to streamline your test code.
```

```
#include <stdio.h>
#include <stdlib.h>
#include "json.h"
#include "miniunit.h"
```

```
// helper headers you wish to use from json.h
```

```
void _bst_insert(BSTNode** a_root, char* key, Element element);
```

```
// all tests from JSON 4, do not delete old tests!!!
```

```
static int _test_parse_object_two_values() {
    mu_start();
    //-----
```

```
    char const* input = "{\"key1\": 123, \"key2\": \"value\"}ABC";
    char const* pos = input;
    BSTNode* root;
    bool is_success = parse_object(&root, &pos);
    mu_check(is_success); // because the input is valid
    // I was too lazy to count the characters, so made C count it for me
    // the 3 is the trailing characters
    mu_check(pos == input + strlen(input) - 3);
    mu_check(root != NULL);
    // we could check specific locations in the tree,
    // or we could just use the helper which does not care
    Element* a_element = get_element(root, "key1");
    mu_check(a_element != NULL);
    mu_check(a_element->type == ELEMENT_INT);
    mu_check(a_element->as_int == 123);
    element = get_element(root, "key2");
    mu_check(a_element != NULL);
    mu_check(a_element->type == ELEMENT_STRING);
    mu_check_strings_equal(a_element->as_string, "value");
```

```
    free_element((Element){.as_object = root, .type = ELEMENT_OBJECT});
    //-----
```

```
mu_end();  
}  
  
int main(int argc, char* argv[]) {  
    mu_run(_test_parse_object_two_values);  
    mu_run(_test_write_int_zero);  
    return EXIT_SUCCESS;  
}
```

2. That string looks super messy, is there a nicer way we can write it?

You may find it easier to create JSON objects as files. This will prevent the need to escape all the double quotes. You could create helper functions to make your tests easier to write. Some examples are below, Example JSON object file contents:

```
{  
    "key1": "This is a string",  
    "key2": 12345,  
    "nested_object": {  
        "with a nested list": [ 1, 2, true, null ],  
        "okay, this object is probably a bit complex": true,  
        "you probably want to test simpler ones :)": false  
    }  
}
```

3. I finished [HW15](#), can I submit this homework without doing objects?

The requirement for this assignment is you must have more functionality working than you had in the previous JSON homework. You do not need to finish all of objects to get some credit, just printing JSON objects would get you some credit.

4. Isn't this requirement unfair for those who finished [HW15](#)?

Those who already finished [HW15](#) already got credit for that code. That said, if you did well on the previous parts you should have no trouble handling objects here.

5. Can I turn in this assignment late?

Due to how close we are to the end of the semester, late work is unlikely to be accepted. Submit early and often and follow TDD to ensure you get as much credit for your work as possible.

Updates

Updates will be posted here if any.

8/5/2022 • Fixed some typos with address fields.