Assignment 4: Into the void*

Due: Mon Oct 30 11:59 pm
Ontime bonus 5%. Grace period for late submissions until Wed Nov 1 11:59 pm

Assignment by Julie Zelenski and Michael Chang

> He who fights monsters might take care lest he thereby becomes a monster. And when you gaze long into a void*, the void* also gazes into you.
> —Friedrich Nietzsche, Beyond Good and Evil

Learning goals

Completing this assignment will level up your understanding and skills with:

- the purpose and use of function pointers in C
- using generic `void*` interfaces as a client
- implementing a generic `void*` function using raw memory operations
- debugging memory errors like a C warrior-ninja!

Overview

Watch video walkthrough! (https://youtu.be/YKq-SP6mgbM)

This assignment consists of some code-study exercises and two small programs to write. For code-study, we will look `scandir`, `bsearch`, and various comparison callback functions. The programs to write are simplified versions of the unix utilities `ls` and `sort`.

Get started

Check out the starter project from your cs107 repo using the command

```
git clone /afs/ir/class/cs107/repos/assign4/$USER assign4
```

The starter project contains `code.c` and `comparison.c` with the code for the code-study exercises, C files `binsert.c`, `myls.c` and `mysort.c`, and the supporting `Makefile`, `custom_tests`, and `readme.txt` files. Our sample solutions and some sample input files are available in the `samples` subdirectory.

1. Code study: scandir

In assign2, you used the `readdir` function to iterate over the entries in the directory. `scandir` is a fancier function for gathering directory entries that is layered on `readdir`. Read its man page (`man scandir`) to be introduced to the function.

A `scandir` implementation (from musl (http://www.musl-libc.org)) is shown below. This function bares the inner soul of C (function pointers, dynamic allocation, triple star, oh my!) but your last three weeks of immersion into deep C waters has made you ready for this. Go slow, draw pictures, and ask questions about anything you don't understand.

```
1  int musl_scandir(const char *path, struct dirent ***res,
2        int (*sel)(const struct dirent *),
3        int (*cmp)(const struct dirent **, const struct dirent **))
4  {
5      DIR *d = opendir(path);
6      struct dirent *de, **names = NULL, **tmp;
7      size_t cnt = 0, len = 0;
8
9      if (!d) return -1;
10
11     while ((de = readdir(d))) {
12         if (sel && !sel(de)) continue;
13         if (cnt >= len) {
14             len = 2*len+1;
15             if (len > SIZE_MAX/sizeof(*names)) break;
16             tmp = realloc(names, len * sizeof(*names));
17             if (!tmp) break;
18             names = tmp;
19         }
20         names[cnt] = malloc(de->d_reclen);
21         if (!names[cnt]) break;
22         memcpy(names[cnt++], de, de->d_reclen);
23     }
24
25     closedir(d);
26
27     if (errno) {
28         if (names) while (cnt-- > 0) free(names[cnt]);
29         free(names);
30         return -1;
31     }
32     if (cmp) qsort(names, cnt, sizeof *names,
32                  (int (*)(const void *, const void *))cmp);
33     *res = names;
34     return cnt;
35 }
```

Answer the following questions in your readme.txt file:

a. Line 12 makes use of `continue`, a C statement you may not have seen before. Use your C reference or web search to get more information. Explain how `continue` operates and what line 12 does in terms of the `scandir` function.
b. The function calls `realloc` on line 16 without having made a previous call to `malloc`. Why is this valid in this case?
c. On line 16, it assigns the return value from `realloc` to tmp and two lines later copies from the pointer from tmp to names. Why does it not just assign directly to names?
d. Line 27 refers to a mysterious `errno` which arises out of nowhere. Read `man 3 errno` to learn more about its purpose and function. If an allocation failure occurred, what will be the value of `errno`? (Hint: read `NOTES` section of the malloc man page)
e. Line 32 is a little hard to parse, but it is applying a typecast to the function pointer being passed to qsort. Try compiling the code both with and without the cast. What warning/error do you get without the cast? (I argue that casting a function pointer is a sketchy thing to do, but I can appreciate why they chose to do so here. Do you agree? You don't need to answer this thought question, but I just wanted to raise the isse for you to consider for yourself)
f. The `scandir` filter function receives its argument as a `const struct dirent *`; the comparison function receives its arguments as `const struct dirent **`. Why the inconsistency?

2. Implement `myls`

You have used `ls` many a time to list a directory's contents, now it will be your turn to put your implementor hat on and write a simplified version of this workhorse unix utility.

The `myls` program operates similarly to standard `ls` but with many simplifications and some differences. While it may be helpful to think of `myls` as the same as standard `ls` in spirit, please don't mistakenly attempt to match the more complex features of standard `ls`. The list below enumerates the required features for `myls`. If in doubt about the expected behavior for `myls`, your best recourse is to observe the behavior of the sample solution rather than compare to standard `ls`.

- `myls` takes zero or more directory paths as arguments. It lists the directory entries from each path. If no paths are given, `myls` prints the entries from the current directory (`.`)
- `myls` only supports directory paths as arguments. If an argument refers to a file or non-existent path, an error message is printed and that argument is skipped.
- `myls` prints the entries one per-line.
- `myls` ignores entries whose names start with `.` unless invoked with the `-a` flag
- `myls` adds a trailing slash when printing the name of an entry that is itself a directory.
- `myls` prints the entries in a directory sorted in order by name. Names are compared case-insensitively (e.g. `strcasecmp`). If invoked with the `-z` flag, the output is sorted to list entries that are directories first, followed by non-directories. Within the group of directories/non-directories, entries are sorted by name.
- `myls` supports no command-line flags other than `-a` and `-z`

Read ahead to the section on "Comment starter code" for guidance on reviewing the initial `myls.c` program. `myls` should call the standard `scandir` function and not re-implement its functionality. This little program is a nice warmup for writing and using function pointers!

3. Code study: bsearch

Writing a fully correct binary search is notoriously difficult. (A good read: Are you one of the 10% of programmers who can write a binary search? (https://reprog.wordpress.com/2010/04/19/are-you-one-of-the-10-percent/)) Having to write as a generic `void*` function only adds to the challenge. Below is Apple's code for `bsearch` , which I find to be a fairly tight and reasonably readable implementation.

```
void *apple_bsearch(const void *key, const void *base, size_t nmemb,
               size_t width, int (*compar)(const void *, const void *))
{
    for (size_t nremain = nmemb; nremain != 0; nremain >>= 1) {
        void *p = (char *)base + (nremain >> 1) * width;
        int sign = compar(key, p);
        if (sign == 0)
            return p;
        if (sign > 0) {  /* key > p: move right */
            base = (char *)p + width;
            nremain--;
        }       /* else move left */
    }
    return NULL;
}
```

One detail I want to draw your attention to is the treatment of a `void*` within a generic function. It is not legal to dereference or do pointer arithmetic on a `void*` , thus a generic function must cast from `void*` to `char*` to perform those operations. In exercise 1 of lab4 (/class/cs107/lab4), you reviewed `musl_memmove` , which copied its `void*` arguments into local variables of `char*` at the start of the function and thereby avoided the need for further casting. In contrast, `apple_bsearch` maintains its arguments as `void*` and explicitly casts before each pointer arithmetic. My preference is for this latter strategy. The `void*` type communicates the special nature of this pointer, rather than misleadingly labeling it as an ordinary c-string. Keeping it a `void*` also means the compiler will squawk should you accidentally dereference or apply pointer arithmetic. An expression that manipulates the `void*` requires an explicit cast. That typecast confirms the programmer's deliberate intent and serves as documentation. Remember that a typecast is a completely compile-time operation, it incurs no runtime performance cost.

Answer the following questions in your readme.txt file:

a. For problem 2 of assign1 (/class/cs107/assign1/), we read Joshua Bloch's article Nearly All Binary Searches and Mergesorts are Broken (https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html). Does the `apple_bsearch` exhibit the bug called out in the article? Justify why or why not.
b. Assume the variable `void *arr` is the base address of the array, `void *found` the address of the matching element and `size_t width` the size of each element in bytes. What is the C expression that converts `found` into its corresponding array index?
c. The `bsearch` man page indicates that the client's array should contain elements in ascending sorted order according the comparison function. If the client calls `apple_bsearch` on an array that is unsorted or is sorted relative to a different comparison function, consider the possible consequences. Might it crash or halt the program? Can it return a false positive? (returns pointer to non-matching element) Can it return a false negative (e.g. returns NULL when array contains matching element)? Briefly explain your answer.

4. Code study: void* blues

As we observed in lecture, the necessarily permissive nature of a `void*` interface makes for a treacherous client experience. There are `INT_MAX` ways to misuse a generic function with nary a peep from the compiler about these transgressions. Let's explore this situation further using the code in the `comparison.c` file.

The program contains a correct implementation of a generic `gfind_max` function that finds the maximum element in an array. The `test_max` function makes four calls to `gfind_max`. The first call is completely correct and prints the expected result. Each of the subsequent three calls is incorrect in some way.

a. For each incorrect call to `gfind_max`, work out what is printed and then verify that your understanding is correct by running the program. In your readme.txt, indicate what is printed for each and explain why it is the result from the call. Drawing memory diagrams and/or tracing in gdb may be very helpful in understanding the behavior.

Now examine the function `test_bsearch` also in `comparison.c`. As a rule, for `bsearch` to be able to work properly, the array must be sorted according to the same comparison function that the search is using. (refer to question 3c above). The programmer who wrote this function is confounded by why they couldn't get their code to work using the same comparison function. They eventually got it working by resorting to using a different comparison for search than sort. They know this can't be good, but were unable to identify the correct fix. Time to investigate!

Answer the following questions in your readme.txt:

b. Compile the program as-is and run it to observe that it does seem to work despite the mismatch in comparison functions. Change the code to use `cmp_first_char` as the comparison function for both sort and search. Run this version and it crashes. Where in the code does it crash?
c. The original author's workaround was to add a different comparison function to be used for search. It is a big red flag that this comparison function typecasts its two `void*` arguments to *different* pointee types. The fact that it manages to "work" at all is sketchy in the extreme and depends on a precise detail of how `bsearch` is implemented. What detail is that? If you very carefully read the man page for `bsearch`, you will see that this detail is guaranteed to be true for a conforming implementation, but that still doesn't make it a good idea to depend on it in this way.
d. Identify the proper fix to the code that makes the program work correctly and sort and search both use the same comparison function `cmp_first_char`, as they should.

The point of this exercise is to highlight the necessity of maintaining vigilance as a client of a `void*` interface. It also foreshadows the futility of trying to get the correct code via trial and error. While randomly permuting `* &` and typecasts might eventually land on a correct combination, this approach does absolutely nothing for your understanding. Instead if you take the time to work through the operation on paper, draw diagrams, and trace execution in gdb, you can become confident about what level of indirection is appropriate in what context and why. Ask questions about what you don't understand!

Review and comment starter code

Both `myls.c` and `mysort.c` are given to you with a small amount of code to handle the command-line arguments. Before starting on either program, first read and understand the given code, work out how to incorporate it into your plans, and finally add comments to document your strategy. The starter code for this assignment is intended to help you get going, but you are free to remove/change this code as you prefer.

Some topics to explore as self-test: (do not submit answers)

- Read down into the man page for `readdir` for more details on `struct dirent` and file types.
- Processing command-line arguments is a goopy task. The GNU extension `getopt` helps to clean it up somewhat. Use `man 3 getopt` to learn more about how it works. How is an invalid option detected/reported when using `getopt`?
- Note how a `typedef` is used in `mysort.c` to avoid the crufty syntax of raw function pointer types.
- Do you see anything unexpected or erroneous? *We intend for our code to be bug-free; if you find otherwise, please let us know!*

As per usual, the code we provide has been stripped of its comments and it will be your job to provide the missing documentation.

5. Write binsert

`lfind` and `lsearch` provide two variants of a generic linear search. (These functions are not standard C, but are commonly available, such as on my myth systems). Read `man lsearch` to be introduced to these functions. The feature that distinguishes `lsearch` from `lfind` is that `lsearch` will add the search key to the array if not found. This is a handy convenience!

You are to write the generic function `binsert` which is a `bsearch` variant with this same functionality. A call to `binsert` will perform a binary search to search for the key and if no matching element is found, it will insert the key into the proper position in the sorted array. Consider this function prototype for `binsert`:

```
void *binsert(const void *key, void *base, size_t *nel,
              size_t width, int (*cmp)(const void *, const void *));
```

Specific details of the function's operation:

- The `binsert` arguments are patterned after the arguments to `bsearch` (review `man bsearch`). The one difference is that the number of elements is passed by reference. This is necessary as `binsert` will update the count when inserting a new element into the array.
- If `binsert` does not find a matching element, the key is inserted into the array at the proper position and the number of elements is incremented. It is the client's responsibility to ensure the array has sufficient space to accommodate a new element. `binsert` does no allocation or deallocation of the client's array memory.
- The function returns a pointer to a matching array member or to the newly added member if no existing match was found.
- You should copy/paste the code from `apple_bsearch` into `binsert` to get your starting point. It would be ideal for `binsert` to simply call `bsearch` and not repeat its code, but the standard `bsearch` doesn't expose the necessary information from an unsuccessful search that would allow you to properly position the new element. Thus you would have to make a second search yourself anyway, so code sharing would be nullified.

Write your implementation in the `binsert.c` file. You will write and test this function in isolation, and then use the function later when writing the `mysort` program. You can test your `binsert` function using our provided `test_binsert.c` program. The `test_binsert` program is integrated with sanitycheck.

6. Implement mysort

What does the `sort` command do?

The unix `sort` command is another example of a filter program like `uniq` and `tail`.

`sort` reads input line-by-line and then prints out the lines in sorted order. The default sort order is lexicographic, but can be changed with various command-line flags. Consider the following sample uses of `sort`:

```
myth> cat samples/colors
red
green
green
red
blue
blue
blue
red

myth> sort samples/colors
blue
blue
blue
green
green
red
red
red

myth> sort -u -r samples/colors
red
green
blue
```

How does `mysort` operate?

The `mysort` program operates similarly to standard `sort` but with many simplifications and some differences. While it may be helpful to think of `mysort` as the same as standard `sort` in spirit, please don't mistakenly attempt to match the more complex features of standard `sort`. If in doubt about the expected behavior for `mysort`, your best recourse is to observe the behavior of the sample solution rather than compare to standard `sort`.

Here are the required features for `mysort`:

- `mysort` reads one file; either the named file (if specified as argument) or standard input (if not).
- The default sort order for `mysort` is case-sensitive lexicographic order (e.g. `strcmp`).
- If invoked with `-l` flag, `mysort` instead sorts by line length. Lines of the same length are sorted lexicographically.
- If invoked with the `-n` flag, `mysort` instead sorts by string numerical value (applies `atoi` to each line and compares numbers).
- If invoked with the `-r` flag, the sort order for `mysort` is reversed.
- If invoked with the `-u` flag, `mysort` discards duplicate lines during read, thus the sorted output contains only the uniqe lines from the input. The sort comparator is used to determine which lines are duplicates.
- The flags to `mysort` may be used alone or in combination. If both `-l` and `-n` used, whichever flag is last on the command line "wins" as sort order.
- `myls` supports no command-line flags other than `-l -n -r -u`.

Requirements that apply to the internal implementation of `mysort`:

- `mysort` reads a line into a stack array using `fgets`. This stack array should be sized to a large maximum size (see `MAX_LINE_LEN` constant). We will not test on inputs containing any lines longer than this maximum. You may furthermore assume that all inputs will end with a final newline. This avoids having to make any special case out of the last line in the input.
- After reading a line you intend to store, it should be copied to dynamically-allocated storage of the appropriate size. The function `strdup` will be handy here.
- `mysort` should be able to handle an input containing any number of lines. Such a large array of lines could much too big for the stack, so this array must be heap-allocated. Because the number of lines cannot be determined in advance, `mysort` should allocate the array to a minimum initial number (see `MIN_NLINES` constant) and then each time it fills up, double the size.
- The `-u` option of `mysort` must call your `binsert` function to sort and unique lines as you read from the input. It should not read/store duplicates for later removal.

One of the key goals for `mysort` is to cleanly handle the mix of sorting options without repeating code or overly complicating things. This requires thoughtful design and you may have to iterate a little until you arrive at a pleasing end result.

Advice/FAQ

Don't miss out on the good stuff in our companion document!

Go to advice/FAQ page (advice.html)

Grading

Here is the tentative point breakdown:

Readme questions (35 points)

- **readme.txt**. (35 points) For the code-study questions, you will be graded on the understanding of the issues demonstrated by your answers and the correctness of your conclusions.

Functionality (85 points)

- **Sanity cases** (25 points) Correct results on the default sanity check tests.
- **Comprehensive/stress cases** (25 points) Correct results for additional test cases with broad, comprehensive coverage and larger, more complex inputs.
- **Clean compile** (2 points) Compiles cleanly with no warnings.
- **Clean runs under valgrind** (15 points) Clean memory reports for all programs when run under valgrind. Memory errors (invalid read/write, use of freed memory, etc) are significant deductions. Memory leaks are a minor deduction. Every normal execution path is expected to run cleanly with no memory errors nor leaks reported. We will not test exceptional/error cases under Valgrind.

- **Reasonable efficiency** (15 points) We expect programs to be reasonably efficient in use of time and space. Full points are awarded for being on par (2-3x) with the sample program; deductions will apply for immoderate use of memory/time. There is no bonus for outperforming the benchmark (and efforts to do so could detract from your code quality...).

Code review (buckets together weighted to contribute ~20 points)

- *Use of pointers and memory*. We expect you to show proficiency in handling pointers/memory, as demonstrated by appropriate use of stack versus heap allocation, no unnecessary levels of indirection, correct use of pointee types and typecasts, const correctness, and so on.
- *Program design*. We expect your code to show thoughtful design and appropriate decomposition. Data should be logically structured and accessed. Control flow should be clear and direct. When you need the same code in more than one place, you should unify, not copy and paste. If the C library provides functionality needed for a task, you should leverage these library functions rather than re-implement that functionality.
- *Style and readability*. We expect your code to be clean and readable. We will look for descriptive names, defined constants (not magic numbers!), and consistent layout. Be sure to use the most clear and direct C syntax and constructs available to you.
- *Documentation*. You are to document both the code you wrote and what we provided. We expect program overview and per-function comments that explain the overall design along with sparing use of inline comments to draw attention to noteworthy details or shed light on a dense or obscure passage. The audience for the comments is your C-savvy peer.

On-time bonus (+5%)

The on-time bonus for this assignment is 5%. Submissions received by the due date earn the on-time bonus. The bonus is calculated as a percentage of the point score earned by the submission.

Finish and submit

Review the How to Submit (/class/cs107/submit.html) page for instructions. Submissions received by the due date receive the on-time bonus. If you miss the due date, late work may be submitted during the grace period without penalty. No submissions will be accepted after the grace period ends, please plan accordingly!

How did it go for you? Review the post-task self-check (/class/cs107/selfcheck.html#assign4).

Further reading

I highly recommend you check out: Engineering a sort function (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.8162&rep=rep1&type=pdf) This article is a great read on the thought and engineering that went into the qsort library function. Jon Bentley and Doug McIlroy are two Hall of Famers from the former Bell Labs and anything they write is worth reading!