Lab 4: void * and function pointers

Lab sessions Tue Oct 24 to Thu Oct 26

Lab written by Julie Zelenski

> Find another world where freedom waits, past the the stars in fields of ancient void
> —Black Sabbath, *Into the Void*

Learning goals

During this lab, you will:

- explore how C void*/function pointers support generic functions
- practice writing callback functions
- implement a generic function that operates on data of any type

First things first! Find an open computer to share with a partner. Introduce yourself and tell them about your plans for Halloween.

Get started

Clone the repo by using the command below to create a lab4 directory containing the project files.

```
git clone /afs/ir/class/cs107/repos/lab4/shared lab4
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab4).

Lab exercises

1) Code study

memmove

The C library provides a handful of raw memory routines (e.g. `memcpy`, `memset`, ...) that operate on data of unspecified type. Let's take a look inside `memmove` (version below from musl (http://www.musl-libc.org)) to better understand how these kind of functions are implemented.

```
1   void *musl_memmove(void *dest, const void *src, size_t n)
2   {
3       char *d = dest;
4       const char *s = src;
5
6       if (d==s) return d;
7       if (s+n <= d || d+n <= s) return memcpy(d, s, n);
8
9       if (d<s) {
10         for (; n; n--)
11              *d++ = *s++;
12      } else {
13          while (n) {
14              n--;
15              d[n] = s[n];
16         }
17      }
18      return dest;
19  }
```

Go over the code with your partner and discuss these questions:

a. The function's interface declares its parameters as `void*` pointers, but internally it manipulates these pointers as `char*`. Why the inconsistency? What would be the consequence of trying to reconcile the discrepancy by declaring the interface as `char*` or changing the implementation to use `void*`?
b. Note that there is no typecast on lines 3 and 4 when assigning from an untyped pointer to a typed pointer. A `void*` is the universal donor/recipient and can be freely exchanged with other pointer types, no cast necessary. I find the author's decision to not cast

pleasing, given my preference for casting only where you absolutely must. As I mentioned in lecture, the internets argue about this endlessly (https://stackoverflow.com/questions/605845/do-i-cast-the-result-of-malloc) if you want to hear more about why it is contentious.

c. What special case is being handled on line 6?
d. What special case is being handled on line 7? What is the difference between the functions `memcpy` and `memmove`? (Check the man page for information)
e. What two cases are being divided by the if/else on Lines 9/12? Why are both cases necessary?
f. Despite lines 10-11 and lines 13-15 both accomplishing a similar function (i.e. copy bytes from source to destination), they are oddly dissimilar in construction: `for` instead of `while`, pointer arithmetic/dereference versus array indexing. Which version do you find easier to follow? Which one is easier for you to verify correctness? Do you think it would be better to write both in same style?
g. Trace the call `musl_memmove(NULL, "cs107", 0)`. Will it result in a segmentation fault from trying to read/write an invalid pointer? Why or why not? What about the call `musl_memmove(NULL, "cs107", -1)`? Verify your understanding by running the `code` program.
h. Use `make clean` and `make` to see the compiler warnings from the code.c program. The complaints you get are warnings targeted at certain invalid calls to the standard `memmove` / `memcpy` functions. The raw memory functions are necessarily defined with an extremely permissive interface (any two pointers and size will do) which makes them ripe for misuse. These warnings catch only a few easily-identified errors, there are many more errors that won't be reported, but it's interesting to note that gcc thought it important enough to make a special case out of them.

The implementation of `memmove` should remind you of the `strncpy` function you studied back in lab2 (/class/cs107/lab2). The `memxxx` functions have much in common with their `strxxx` equivalents, just without the special case to stop at a null byte. In fact, the `memxxx` functions are declared as part of the `<string.h>` module and quite possibly written by the same author.

Comparison functions

The sort/search functions of the C standard library are written as generics (e.g. using `void*`) and require the client to supply a callback comparison function to compare elements. A comparison function often only needs simple logic to do its task, but managing the syntax and applying the correct level of indirection is where the trickiness comes in.

The client's comparison function must fit the standard comparison function (https://www.gnu.org/software/libc/manual/html_node/Comparison-Functions.html#Comparison-Functions) as described in the GNU libc manual. I lightly paraphrased its example comparison function below:

```
int compare_doubles(const void *p, const void *q)
{
    double first = *(const double *)p;
    double second = *(const double *)q;

    return (first > second) - (first < second);
}
```

One essential feature to note is that the arguments to a comparison function are two **pointers to the elements** to be compared, **not the values** of those elements. As a rule, the first thing a callback function should do is to cast the incoming `void*` arguments and copy into the variables of the proper type. The rest of the function can then operate on those properly-typed variables and receive the benefit of the compiler's help in respecting type safety, which you do not get if directly accessing the `void*`.

A more minor detail is that it is only the sign of the comparison function result that matters. For example, -1 or any negative result from the comparison function is taken as an indication that the first element is less than the second.

Look back at the `compare_doubles` function above. Work out these questions with your partner:

a. What is the result of evaluating any inequality such as `first > second`? (Try it in gdb to confirm!) Interesting, the relational/equality operators are defined to produce a value of either 0 or 1 and "clever" programmers are wont to take advantage of this.
b. How does the subtraction of the two inequalities compute a result of the correct sign?
c. The more obvious approach would be a chained if/else of three separate cases for less/equal/greater. That version would certainly be more readable and easier to verify

correct. What do you think might have motivated the author instead to write it as shown above?

Comparing ints

Your colleague checks in a comparison function for integer elements into your team's repo. The implementation subtracts the two values and uses their difference as the comparison result.

```
int cmp_int(const void *p, const void *q)
{
    int first = *(const int *)p;
    int second = *(const int *)q;
    return first - second;
}
```

During code review, you point out to your colleague that this function will not work correctly in case of overflow. For example, if the first element is a large enough positive value and second a large enough negative value such that their difference exceeds `INT_MAX`, the function result will be incorrect, causing those two elements to be incorrectly ordered.

a. Use the `ints` program to observe the consequence of the overflow bug. The test program puts a few extreme values in the array (to deliberately trigger overflow) and randomizes the remaining array elements. It qsorts the array using the above comparison function. Run the program several times. You should observe that the extreme values end up in wacky places, while the other elements are mostly sorted correctly.

Your colleague concurs with your bug report when you demonstrate this reproducible failure. To fix, he proposes to avoid the overflow by promoting the values to the larger bitwidth `long` type before the subtraction:

```
int nobetter_cmp_int(const void *p, const void *q)
{
    long first = *(const int *)p;
    long second = *(const int *)q;
    return first - second;
}
```

Discuss with your partner:

b. Change the `ints` program to use this version of the comparison function and try running it several times. This proposed fix does absolutely nothing in terms of correcting the error. Why not?

Comparing structs

Let's do a quick recap of structs before our next example: C struct declarations are almost, but not exactly, the same as C++. In C, the following declares a new struct type:

```
struct coord {
   int x, y;
};
```

The name of the type is `struct coord` and you cannot drop the struct keyword; declaring a variable of type `coord` will not compile.

The `.` (dot) operator is used to access the fields within a struct. If you have a pointer to a struct, your first attempt to access the fields is likely to run afoul of the fact that `.` has higher precedence than `*`. You can add parentheses to force the desired precedence, or better, use the `->` operator which combines `.` and `*` for this common need.

```
struct coord origin;                          // struct on stack
struct coord *p = malloc(sizeof(struct coord)); // struct in heap
                  // (note: sizeof works correctly for structs)

origin.x = 0;  // access field from struct variable

               // these next 3 lines access field via struct pointer
*p.x = 0;      // WRONG! precedence applies . first then *
(*p).x = 0;    // OK: parens used to override precedence
p->x = 0;      // BEST: preferred way to access
```

The `searchsort.c` file in the starter code contains the example sort-search program (https://www.gnu.org/software/libc/manual/html_node/Search_002fSort-Example.html#Search_002fSort-Example) from the GNU libc manual. The example demonstrates using the `qsort` and `bsearch` library functions on an array consisting of elements of struct type.

a. Review the program to see a struct definition and its use. Note how the generic functions can support this new type by virtue of the client-supplied comparison function. The comparison function orders the struct critters by comparing the name field. Two critters of the same name would be considered equal by the function.
b. As an exercise in working with callback functions, edit the comparison function to break ties between critters of the same name by comparing their species.

2) gdb protip: printing arrays

One handy gdb feature we want to get into your repertoire is how to print arrays. If you print a stack array from within the function it is declared, gdb will show the array and its contents. In that context, gdb has access to both the element type and the count of elements, and uses it to print a nice representation of the entire array. However it cannot automatically do the same in other contexts, such as for a heap array or for an array/pointer passed into a function. It is possible to print the entire array in those contexts, but you have to provide more information to gdb. Let's see how!

a. Run gdb on the program `ints`. Set a breakpoint on `main` and step into the function past the variable declaration/initializations.
b. Try `p nums`. Here in the context of its declaration, gdb knows that it is a stack array of a certain size and can show the entire stack array. Great!
c. Now try `p argv`. All gdb knows about argv is that it is a pointer. Bummer.
d. Try `p argv[0]@argc` and gdb will now print the entire contents of the argv array. Hooray!
e. The syntax to learn is `p ELEM@COUNT` where `ELEM` is the 0th element and `COUNT` is the count of elements to print. ELEM and COUNT are C expressions and can refer to any variables in current scope. Try `p nums[1]@2` to show a 2-element portion in the middle of the array.
f. You can also add in a typecast if needed. For example, given a parameter `ptr` of type `void*` that you know is the base address of an array of `nelems` elements of type `char*`, you could print the entire array as `p *(char **)ptr@nelems`.

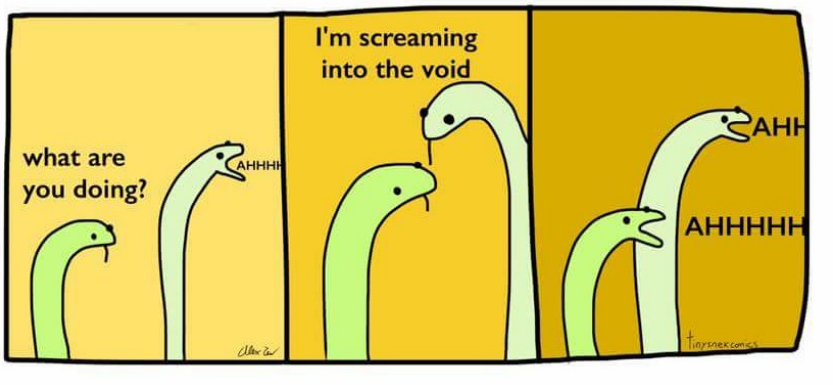
3) Write, test, debug, repeat

Now it's your turn! The `dups.c` program is intended to process an array of numbers given as command-line arguments and report whether the array contains any duplicate values and then remove those duplicates.

a. Read over the given code to see what you start with. The `has_duplicates` function is correctly implemented and models the standard technique to iterate over a generic array and call the client's callback function. Be sure you understand this code before going further!
b. The program is missing the implementation of the integer comparison function. The program intends to use a `cmp_magnitude` function that compares elements by their absolute value, i.e. -5 is > 0, -7 == 7, and 1 < -2. Rather than start with the buggy int comparison looked at earlier in this lab, write your own simple version from scratch that returns a correctly signed result based on comparing the absolute value of the two elements.
c. The program is also missing the implementation of the `remove_duplicates` function. Use the approach shown in `has_duplicates` to find duplicate elements within the generic array and upon finding one, remove it from the array by moving the neighboring elements down by one position and decrementing the array's element count. The entire block of elements should be moved in one call; this is much more efficient than a loop that copies the elements one by one. You must use `memmove`, not `memcpy`, in this situation — why?
d. Using sanitycheck to verify the correctness of your work.

Check off with TA

Before you leave, complete your checkoff form and ask your lab TA to approve it so you are properly credited. If you don't complete all the exercises during the lab period, we encourage you to followup and finish the remainder on your own. Try our self-check (/class/cs107/selfcheck.html#lab4) to reflect on what you've done and how it's going.


what are you doing?
AHHHH
I'm screaming into the void
AHH
AHHHHH