C Language Laboratory

CS 105, Spring 2018

Due on Tuesday, January 30, 2018, at 11:59 PM

This laboratory exercise gives you practice programming in the C language and demonstrates how data are stored in bytes and words. It will introduce you to the "C mindset," which may be significantly different from the way you are used to thinking about programming.

In many ways, the C language is like Java. The syntax for variable and function declarations, assignment statements, `for` and `while` loops, and `if`-statements are the same in both languages. The big difference is that in C we have a different view of data, one that is closer to the actual hardware. We must be aware of where in memory values are located and how much space they occupy. The exercises in this laboratory assignment will give you practice in thinking about variables, pointers, and arrays—and how they relate to addresses and values in memory.

As with the previous lab, work in teams of two. To get started, download the file `clang.tar` from the course web page. Change to a protected directory and unpack the file with the command

```
% tar xvf clang.tar
```

(A note on color-coding: In a few cases we display commands that you type in a terminal window and the resulting output. We use % for the prompt. The characters that you are to type are in green, and the system's responses are in purple.)

You will now have a directory `clang` which contains this writeup, a `Makefile`, and three C language programs. As you follow the instructions below, you will produce or modify three files.

```
strings.solution
arrays.c
lists.c
```

Be sure to put the names of all the team members at the top of the C language files and at the bottom of `strings.solution`. Submit all three files—as separate files, not zipped or tarred—on the course submission page. Use all the team members' names when submitting.

A few paragraphs below, labeled "Reflections," raise relevant questions and direct you to important points that we want you to learn from the exercises. Remember to return to them when you have the opportunity to contemplate.

1 Re-interpreting Data Values

The program `strings.c` reads six integers into an array. It then interprets the first four integers as a string and the last two as a `double` and prints the results. Your job is to find integers that will cause the program to print some specified values.

Begin by compiling the program. We have given you a `Makefile`, so you can just type

```
% make strings
```

on the command line. Do not change the program's source. Next, create a text file named `strings.solution` with six integers, one to a line. Begin by making them all zero. Put your names on the seventh and eighth lines. Run the `strings` program with the input redirected from the `strings.solution` file.

```
% ./strings <strings.solution

0.0000000000000000
```

The blank line in the output shows that the string is empty. The `double` is zero. As a further warm-up, change the first of the six integers to 14132. The string now has two characters—which happen to be digits. (It is a string of characters, not a number!) The `double` is still zero.

```
% ./strings <strings.solution
47
0.0000000000000000
```

Your task Fill `strings.solution` with six integers to produce this result.

```
% ./strings <strings.solution
Cecil Sagehen
3.1415926535897931
```

When you have the solution, submit the file `strings.solution` on the course submission page. Remember to put all team members' names in the file, *after* the six integers.

Hints and suggestions It is possible, but long and tedious, to compute by hand the four integers corresponding to "Cecil Sagehen." But it is not practical (or a good use of your lifespan!) to compute the two integers corresponding to the decimal expansion of π. Think about writing a short program, separate from `strings.c`, that will calculate the integers for you.

Reflections The actual values of the six integers are not important. If they were all you cared about, you could ask someone in the lab. Be sure that you understand what is happening with the bytes in memory. Also, take some time to understand the pointer arithmetic and type casts in the source file `strings.c`.

The six-integer sequence you produced is not unique. Other sequences will produce the same result. How many different solutions are there?

2 Re-shaping Arrays

This part of the lab is an exploration into how arrays are stored in memory. Suppose that we have a two-dimensional 4×7 array `tda` of integers whose values encode the indices. That is, `tda[i][j]` has the value $10i + j$. If we print the array row-by-row we obtain this:

```
00 01 02 03 04 05 06
10 11 12 13 14 15 16
20 21 22 23 24 25 26
30 31 32 33 34 35 36
```

Remember that arrays are stored in row-major order, so in memory the array looks like this:

```
00 01 02 03 04 05 06 10 11 12 13 14 15 16 20 21 22 23 24 25 26 30 31 32 33 34 35 36
```

If we take that block of memory—without changing the values stored there—and consider it as a 7×4 array, we get this:

```
00 01 02 03
04 05 06 10
11 12 13 14
15 16 20 21
22 23 24 25
26 30 31 32
33 34 35 36
```

The same memory can be considered a one-dimensional 28-element array, a two-dimensional 4×7 array, a two-dimensional 7×4 array, a three-dimensional $7 \times 2 \times 2$ array, and so on.

As we saw in class, there is another way to represent two-dimensional arrays—as arrays of arrays. The idea is to have a one-dimensional array of *rows*. Each element in that array is a pointer that points to the beginning of another one-dimensional array which contains the data values of that particular row. With this representation, the 7×4 array shown above would be an array of seven pointers, each pointing to an array of four integers. The strategy uses a little more memory (for the pointers), but it is more flexible. The rows need not all be the same size.

Your task Your lab material contains an almost-complete program `arrays.c`. It declares and initializes a 4×7 array `tda` like the one above. It also declares an array `aoa` of seven rows. Your job is to fill in seven assignment statements in the `main` function so that `aoa` is a 7×4 array like the one above, except with the rows in reverse order.

Make no changes to the program, except to add your names and complete the assignment statements. When you are ready, type

```
% make arrays
% ./arrays
```

If your assignments are correct, you will see the output shown in Figure 1., showing the original array and the modified one. Submit your program `arrays.c` on the course submission page.

Hints and suggestions Avoid trial and error. Think about the starting points of the various rows of the result, relative to the beginning of `tda`.

```
% ./arrays
00 01 02 03 04 05 06
10 11 12 13 14 15 16
20 21 22 23 24 25 26
30 31 32 33 34 35 36

33 34 35 36
26 30 31 32
22 23 24 25
15 16 20 21
11 12 13 14
04 05 06 10
00 01 02 03
```

Figure 1: The correct output for the program `arrays`.

Reflections Notice that the assignments to `aoa` are made *before* `tda` is initialized. Why does that work? Notice also that the bodies of `print_two_dim_array` and `print_array_of_arrays` are character-for-character *identical.* Why is it necessary to have two functions? Be sure that you understand the difference between the two ways of representing two-dimensional arrays.

3 A Linked List

For this part of the lab, you will complete a simple linked list program. It is the sort of exercise that you may have done in a data structures course. We have given you a partially-complete program `lists.c`. You are to implement three functions with these prototypes:

```
void makeempty(cell_t** thelist)
void prepend(int newvalue, cell_t** thelist)
void reverse(cell_t** thelist)
```

The type `cell_t` is a structure with two members, `value` and `next`. For us, a *list* is a pointer to a `cell_t`. The `next` field in the structure, a pointer, is *the rest of the list.* The empty list is a pointer whose value is `NULL`.

All three functions operate on a list. The function `makeempty` removes and recycles all the elements of the given list. The function `prepend` creates a new `cell_t` with the specified value and places it at the front of the list. The function `reverse` reorders the elements in the list. It does so by moving the elements of the list, not by creating new copies of elements.

Your task Implement the four functions listed above. Do not change the functions `printlist` and `main` When you have finished, compile and run the program.

```
% make lists
% ./lists
```

Make sure that the output is correct by comparing it to the listing in Figure 2. Submit your program `lists.c` on the course submission page.

```
% ./lists
backward
9, 0x971130
8, 0x971110
7, 0x9710f0
6, 0x9710d0
5, 0x9710b0
4, 0x971090
3, 0x971070
2, 0x971050
1, 0x971030
0, 0x971010
backward reversed
0, 0x971010
1, 0x971030
2, 0x971050
3, 0x971070
4, 0x971090
5, 0x9710b0
6, 0x9710d0
7, 0x9710f0
8, 0x971110
9, 0x971130
empty
forward
0, 0x971130
1, 0x971110
2, 0x9710f0
3, 0x9710d0
4, 0x9710b0
5, 0x971090
6, 0x971070
7, 0x971050
8, 0x971030
9, 0x971010
forward reversed
9, 0x971010
8, 0x971030
7, 0x971050
6, 0x971070
5, 0x971090
4, 0x9710b0
3, 0x9710d0
2, 0x9710f0
1, 0x971110
0, 0x971130
empty again
```

Figure 2: The correct output for the program `lists`. The addresses, the hexadecimal values after the commas, may differ in your output, but they should be spaced apart by the same amounts.

What you need to know Pointers are used to refer to elements in the list. Remember that a pointer simply holds an address in memory. If you want it to point to something useful, you must allocate space in memory and set the pointer to the address of that space. Here is how to create an element for a list:

```
cell_t *p = (cell_t *) malloc(sizeof(cell_t));
```

You can then initialize the fields of the list element by assigning to `p->value` and `p->next`. Keep in mind the distinction that `p` is the address in memory of an element and `*p` is the element itself. You will need to allocate space for new elements in `prepend` and `append`.

When an element is created with `malloc`, it lasts until it is explicitly disposed, or until the program ends. To dispose of an element and recycle the memory that has been allocated to it, make a call to `free` with a pointer to the element.

```
free(p);
```

The value of the pointer must have come from a call to `malloc`. There should be only one call to `free` for each call to `malloc`. In `makeempty`, you will need to `free` all the elements in the given list.

Pay special attention to the list argument in the functions you write. A list for us is a pointer to `cell_t`. The argument to your functions is a *pointer to a list,* of type `cell_t**`. In the function

```
void makeempty(cell_t** thelist)
```

the variable `thelist` is a pointer to a list. That is, it is the address of a place in memory that holds the address of the first element of the list. The reason for using a double pointer is so that the function `makeempty` can change the value of the list back in the caller. The last act of `makeempty` (after it has recycled the memory of all the original list elements) is to make the assignment

```
*thelist = NULL;
```

so that the caller's list will now be empty.

As an extended example, we have given you the C code for the function `append`. It creates a new `cell_t` with the specified value and places it at the end of the specified list. Figure 3 shows a heavily-annotated copy of the function.

Hints and suggestions Do not get carried away! You are to write four functions, and each one will be no more than 15 lines long, often shorter. On the other hand, programming with pointers is delicate work. The few lines of code that you write must be precisely correct.

Reflections The function `printlist` prints the addresses of the various list elements. One reason for that is to be able to check that `reverse` creates a list with *the very same elements,* just in a different order. You can use those addresses to determine how many bytes are used for each `cell_t`. How many bytes are actually required by the data in the structure? How many bytes are needed according to our alignment rules? How many bytes are actually used by the system? (The answers are all different.)

```
void append(int newvalue, cell_t** thelist) {
  // Create a new cell to be added to the list.
  cell_t *newelt  = (cell_t*) malloc(sizeof(cell_t));

  // Fill in the components of the new cell.
  // The cell will go at the end of the list, so
  // the next field is NULL.
  newelt->value = newvalue;
  newelt->next = NULL;

  // Appending to an empty list is a special case.
  if (*thelist == NULL)
    *thelist = newelt;

  // If the list is not empty, find the last element,
  // and then tack on the new cell.
  else {
    cell_t *p = *thelist;
    while (p->next != NULL)
      p = p->next;
    p->next = newelt;
  }
}
```

Figure 3: An annotated version of append.