C track: assignment 6

Goals

In this assignment you will write a sorting program similar to the program from lab 3, except that it uses linked lists instead of arrays to hold the list of numbers read in off the command line. In the process, you will learn about one of the fundamental constructs in the C language (structs) and more about pointers.

Language concepts covered this week

- structs

  Structs provide a way to bundle together several related pieces of data into a single piece of data. Structs are commonly used to define new data types.

- typedefs

  Typedefs are a way to give an alias (alternative name) for a type name. Used correctly, they can make code both more concise and more readable.

- linked lists

  Linked lists are a fundamental data type that is constructed using structs and pointers. Linked lists are very common in real code, so it is important that you become comfortable with using them.

Other concepts covered this week

This lab covers sorting in a so-called "functional" style, whereby the original data collection to be sorted is not altered by the sorting process. Rather, the sorting process generates a new, sorted data collection. If you know the Scheme programming language (*e.g.* from CS 1) this style will be familiar to you; if not, it's well worth learning about.

Structs

A struct is a data object which holds other data objects of arbitrary pre-specified types. These data objects are called the "fields" of the struct, and

they are generally conceptually related to each other. For instance, if you wanted a data object to hold the x-y coordinates of a point on a graphical display screen, you could declare a struct to do this:

```
struct point
{
    int x;
    int y;
}
```

This declaration tells the compiler what a `point` is, but doesn't create any points. To do that you do this:

```
struct point p;  /* Creates a new point called 'p'. */
```

However, most C programmers use the following shortcut. In C you can give a type name an alias (alternative name) by using a *typedef*:

```
typedef struct point
{
    int x;
    int y;
} Point;
```

This defines `struct point` as above, and also declares `Point` to be an alias for `struct point`. Now you can create a new `Point` like this:

```
Point p;
```

Note that `Point` and `point` are not the same thing, because you don't use the `struct point` syntax when you're declaring a `Point`. In fact, writing `struct Point` is invalid; you have to just write `Point`. However, a `Point` object is still a `struct`.

How do you use a struct? You can access the field values in two different ways. The first way is to use the `.` operator:

```
int z;
Point p;

p.x = 10;  /* Set 'x' field of p to 10 */
p.y = 20;  /* Set 'y' field of p to 20 */

/* You can use 'p.x' syntax on either side of an assignment statement. */
z = p.x + p.y;  /* = 30 */
```

The second way is used with pointers to structs. Then you would normally use the arrow (`->`) operator:

```
int z;
Point p;
Point *q;  /* pointer to a Point struct */
```

```
q = &p;    /* Now q points to p. */

q->x = 10;  /* Set 'x' field to 10 */
q->y = 20;  /* Set 'y' field  to 20 */

/* You can use 'q->x' syntax on either side of an assignment statement. */
z = q->x + q->y;  /* = 30 */
```

Note that `q->x` is nothing more than syntactic sugar for `(*q).x`.

Recursively-defined structs and singly-linked lists

This lab will feature a new data type called a *singly-linked list*. This is a very flexible data type; whole computer languages (*e.g.* Lisp and Scheme) have been built around it. We will restrict ourselves to a singly-linked list of integers. In C we only need one struct for this, which we'll call a "node":

```
typedef
struct _node
{
  int data;             /* Each node stores a number */
  struct _node *next;   /* and a pointer to the next node in the list. */
} node;
```

Actually, we are declaring a struct called `_node` and `typedef`'ing it to the name `node`. The interesting thing about this struct is that it is recursively defined. A `node` has a `data` field which holds an integer and a pointer to another `node` in its `next` field.

The way we use this is as follows. We allocate memory for a bunch of nodes, then we link the `next` field of one node to another node, link the `next` field of that node to still another node, and so on until all of the nodes are linked together. We set the `next` field of the last node to be equal to `NULL`, which lets us know when we've hit the end. `NULL` is a value `#define`'d in `<stdio.h>` which cannot be the value of a valid pointer (normally it is defined to be zero). The code to create the list might look something like this:

```
int i;
int n_nodes = 10;  /* We want ten nodes. */
node *list;        /* pointer to the list */
node *temp, *prev; /* temporary pointers to nodes */

/* Make the node at the front of the list first. */

list = (node *)malloc(sizeof(node));

/* Check for malloc failure. */
if (list == NULL)  /* malloc failed */
```

```
{
    fprintf(stderr, "Error: malloc() failed.\n");
    exit(1);  /* Give up, abort program. */
}

list->data = 1;
list->next = NULL;  /* No other nodes yet. */

/* Make the rest of the nodes and link them up. */

prev = list;

for (i = 0; i < (n_nodes - 1); i++)
{
    /* Create a new node. */
    temp = (node *)malloc(sizeof(node));

    /* Check for malloc failure. */
    if (temp == NULL)  /* malloc failed */
    {
        fprintf(stderr, "Error: malloc() failed.\n");
        exit(1);  /* Give up, abort program. */
    }

    /* Initialize the new node. */
    temp->data = prev->data + 1;
    temp->next = NULL;

    /* Link previous node to the new node. */
    prev->next = temp;

    /* Set the 'prev' pointer to point to the new node. */
    prev = temp;
}

/*
 * Now, 'list' points to a list of ten nodes,
 * with values 1, 2, 3, ... 10 from front to back.
 */
```

Another way to do this is to create the list starting from the end node and growing it backwards until you hit the first node. This is actually a lot easier:

```
int i, data;
int n_nodes = 10;  /* We want ten nodes. */
node *list;        /* pointer to the list */
node *temp;        /* temporary pointer to node */
list = NULL;       /* NULL represents the empty list. */
data = 10;         /* Starting data value. */

/* Make the nodes and link them up. */

for (i = 0; i < n_nodes; i++)
{
```

```
    /* Create a new node. */
    temp = (node *)malloc(sizeof(node));

    /* Check for malloc failure. */
    if (temp == NULL)  /* malloc failed */
    {
        fprintf(stderr, "Error: malloc() failed.\n");
        exit(1);  /* Give up, abort program. */
    }

    /* Initialize the new node. */
    temp->data = data;
    temp->next = list;

    /* Set the data value for the next node. */
    data--;

    /* Set the 'list' pointer to point to the new node. */
    list = temp;
}

/*
 * Now, 'list' points to a list of ten nodes,
 * with values 1, 2, 3, ... 10 from front to back.
 */
```

You should work through both of these examples until you understand why they work. It's easy to write linked list code which is much more complicated than it needs to be if you're not clear on what's really happening when you create nodes and link them up.

In fact, we've already written all the linked list routines that you'll need for this lab. They are described in the header file linked_list.h and defined in the file linked_list.c. All you have to do is use them. And make sure you **do** use them; we'll take points off for not using them and instead writing the equivalent code out by hand. That's a practice called "reinventing the wheel" which is a bad habit; once someone else has gone to the trouble of writing working, fully-debugged code, it doesn't make sense to rewrite it yourself if their version works fine for your purpose.

Program to write

The program to write for this lab is a variation on lab 3. Recall that in that program, you input a list of numbers and the program printed them out in sorted order. This time you will be doing the same thing, with these differences:

1. You will use a linked list to store the numbers entered on the command line. This will have the advantage that you can enter an arbitrary number of numbers (*i.e.* you're not limited to just 32 numbers).

2. You will use the "quicksort" algorithm to sort the numbers instead of the minimal insertion sort or bubble sort. Quicksort is much more efficient than those algorithms, and is described below.

Your program will be called `quicksorter`. Here is the specification:

1. The program will accept an arbitrary number of numbers on the command line, as long as there is at least one number.

2. If there are no command-line arguments, or if there are arguments but none that represent numbers to be sorted, the program should print out instructions on its use (*i.e.* a usage message). By the way, don't assume that the test script checks this; currently it doesn't, so check it yourself (we will).

3. If any of the arguments to the program are not integers, your program's response is undefined -- it can do anything. (i.e. you shouldn't worry about handling non-integer arguments). The only exception to this is the optional "`-q`" argument to suppress output (see below).

4. The program will sort the numbers using the quicksort algorithm. The numbers on the command line will be stored in a linked list. You should have a separate `quicksort` function to do the sorting. This function should receive exactly **one** argument (the list to be sorted) and return a pointer to a newly-allocated list which contains the same integers as the input list, but in sorted (ascending) order. Do **not** alter the original list in any way. Also, do not pass the length of the list to the `quicksort` function (it isn't necessary anyway).

5. As in lab 3, use `assert` to check that the list is sorted after doing the sort. Use the `is_sorted()` function in `linked_list.c` to check the list. Put the `assert` check at the end of the `quicksort` function, right before it returns the sorted list.

6. Print out the numbers from smallest to largest, one per line.

7. Use the memory leak checker as in the previous assignment to check that there are no memory leaks.

The Sample Output

```
% quicksorter 5 9 -2 150 -95 23 2 5 80
-95
-2
2
5
```

```
5
9
23
80
150
% quicksorter
usage: quicksorter [-q] number1 [number2 ... ]
%
```

Command-line options

Add a "`-q`" command-line argument that suppresses output like you did in assignment 3. Make sure that even when this is in effect, you still sort the numbers. Also, make sure in all cases that you use the `is_sorted()` function in `linked_list.c` (see below) to test whether the sorted list is in fact sorted. Your program should accept single or multiple "`-q`" arguments on the command line (just like lab 3), where multiple "`-q`" arguments are the same as a single one, and if there are no numbers on the command line (*e.g.* just "`-q`" arguments) the program should exit with a usage message.

The quicksort algorithm

Quicksort is a sorting algorithm invented by the computer scientist C. A. R. Hoare. It is much more efficient than bubble sort or insertion sort (technically, it has an O(n log n) average time complexity as compared to O(n^2) for the other two algorithms). Quicksort is usually used to sort an array in place (there is even a C library function called `qsort` which does this), but in this case we are sorting a linked list and we are not modifying the original list, which changes the details of the algorithm somewhat. Quicksort is a recursive algorithm, like the algorithm you used to solve the peg game in lab 4.

Here is the algorithm:

1. If the list has zero nodes (*i.e.* it's empty) or one node, copy the list as-is and return it (it's already sorted). Note that this algorithm works fine on empty lists.

2. Otherwise, create a copy of the first node of the list and "put it aside" for now.

3. Divide the rest of the list into two new lists:

    1. The nodes with values larger than or equal to the value of the first node;
    2. the nodes with value smaller than the value of the first node.

Remember, you have to make these lists yourself and they have to copy the values in the original nodes.

4. Sort these two new lists using a recursive call to the `quicksort` routine.
5. Append everything together in order:
    1. The list of nodes with values smaller than the first node,
    2. the copy of the first node,
    3. and the list of nodes with values larger than or equal to the first node.

    Use the `append_lists` function defined in `linked_list.c` to do the appending.

Memory management

Remember, when you use `malloc` or `calloc` to allocate memory, that memory must be explicitly freed (using the `free` function) before the end of the program or you have a **memory leak**. It can be tricky avoiding memory leaks, but here is a specific suggestion: don't ever do this:

```
node *list;
/* ... some code which assigns values to 'list' ... */
/* Sort the list and set the list pointer to point to the sorted list. */
list = sort_list(list);
```

In this case, `sort_list()` returns a new (sorted) list, and the list pointer `list` is set to the head of that list. This is a memory leak, because the old list that `list` pointed to was never freed. The right way to do this is as follows:

```
node *list;
node *sorted_list;
/* ... some code which assigns values to 'list' ... */
/* Sort the list and set the list pointer to point to the sorted list. */
sorted_list = sort_list(list);
free_list(list);
list = sorted_list;
```

The result is the same, but there's no memory leak.

Again, use the memory leak checker as you did last lab to make sure that you haven't inadvertently leaked any memory. Recall that you do this by putting these lines at the top of the file `quicksorter.c`:

```
#include <stdlib.h>
#include "memcheck.h"
```

and then putting this line in the `main` function before exiting:

```
print_memory_leaks();
```

In addition, if you can't track down a memory leak you might try changing the following line in `memcheck.c` from

```
#define DEBUG 0
```

to

```
#define DEBUG 1
```

and recompiling. This will give more verbose output relating to memory allocation.

Code supplied

There are a number of useful utility functions supplied in linked_list.c and declared in linked_list.h. Please use these instead of writing your own versions; it will save you a lot of time. You will need to `#include "linked_list.h"` in order for this to work.

Testing your program

This is the similar to the test script for assignment 3 (though not as rigorous). Also test your program on some numbers you generate by hand. If you need to use a debugger, read the gdb page to learn how to use `gdb` (the debugger for `gcc`).

Supporting files

- The Makefile.

  Once again, be sure that all the command lines in the Makefile start with tabs, or they will not work.

- The test script.

  Make sure your program passes this before you hand it in.

- linked_list.c and linked_list.h.

- The memory leak checker: memcheck.c and memcheck.h.

To hand in

The `quicksorter` program.

References

- Darnell and Margolis, chapter 9.
- K&R, chapters 1 and 5.
- Any algorithms textbook *e.g.* Sedgewick, Algorithms in C.
