Computation I 5EIA0
Exercise 6: Linked Lists (v1.3 October 5, 2020)
Deadline Wednesday 14 October 23:55

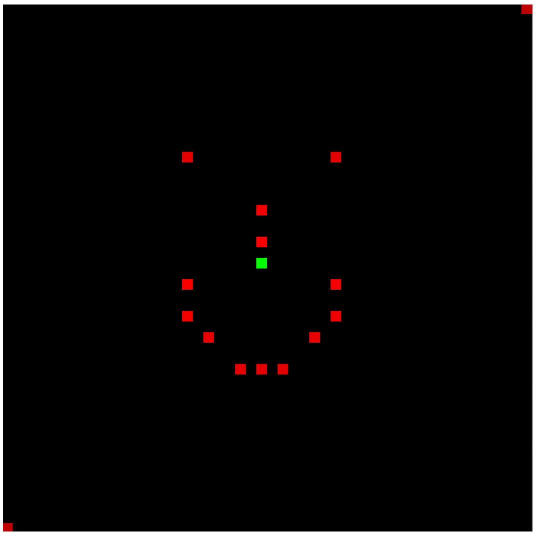

Figure 1: Lists: https://www.youtube.com/watch?v=xqv53866a-4

Task 1. This assignment serves to practice linked lists. You will practise a number of different operations. New nodes can be inserted at the head or at the tail or in the middle of the list. Similarly, nodes can be removed from the head or from the tail or from the middle of the list. Next to that, we will find (and possibly remove) a specific node in the list. At the same time, you'll practise passing linked lists by value and by reference.
In conclusion, the operations that you will implement are:

command	operation
q	quit
p	print list
v	print reVerse list
d	Display list
h	insert at Head
t	insert at Tail
c	show Closest
a	insert After closest
b	insert Before closest
f	remove at Front
e	remove at End
r	Remove closest

The elements of the list are complex numbers, consisting of real (`re`) and imaginary (`im`) `float` number:

```
struct node_t {
  float re, im;
  struct node_t *next;
};
```

In your `main` function declare a linked list of type `struct node_t *`.
Start by implementing the quit command. Print `Bye!` when quitting the program. Print `Unknown command 'X'` when an unknown command is given (with X replaced by the unknown command, of course).

```
Command: x
Unknown command 'x'
Command: q
Bye!
```

Task 2. Implement the 't' command that inserts a complex number at the tail of the list with the `struct node_t *insert_tail (struct node_t *head, float re, float im)` function. Note that the head of the list is passed by value, i.e. the list that's passed to the function is not modified and a new list is returned instead.

Task 3. Implement the 'p' command that prints the list of complex numbers with the `void print_list (struct node_t *head)` function. Print two digits after the decimal point.

```
Command: p
[]
Command: t
re, im? 2 3
Command: p
[2.00+3.00i]
Command: t
re, im? -2 3
Command: p
[2.00+3.00i,-2.00+3.00i]
Command: t
re, im? 4 -5
Command: p
[2.00+3.00i,-2.00+3.00i,4.00-5.00i]
Command: q
Bye!
```

Note that when the imaginary part is negative you must only print the minus sign, and omit the plus sign between the real and imaginary part. The output below is what we do NOT want:

```
[2.00+3.00i,-2.00+3.00i,4.00+-5.00i]
```

Task 4. Implement the 'h' command that inserts a complex number at the head of the list with the `void insert_head (struct node_t **head, float re, float im)` function. Note that the head of the list is passed by reference, i.e. there is no return value and the list that's passed to the function is modified instead.

```
Command: h
re, im? 1 0
Command: p
[1.00+0.00i]
Command: h
re, im? 2 0
Command: p
[2.00+0.00i,1.00+0.00i]
Command: q
Bye!
```

Task 5. We will need to compare two complex numbers, for which we will use the Euclidean distance. Implement the function `float distance (float re1, float im1, float re2, float im2)` that computes the formula:

$$distance = \sqrt{(re_1 - re_2)^2 + (im_1 - im_2)^2}$$

Next implement the 'c' command to compute the complex number in the list that is closest (according to the distance function) to the given number. The function `struct node_t *find_closest (struct node_t *head, float re, float im)` returns the node that's closest to the (re,im) number, and returns `NULL` if the list is empty. If there are multiple numbers that are closest (consider e.g. 0+0i and 2+2i that are both equally close to 1+1i) then return the first one in the list.

```
Command: p
[]
Command: c
re, im? 0 0
No closest node found
Command: h
re, im? 0 0
Command: h
re, im? 2 2
Command: p
[2.00+2.00i,0.00+0.00i]
Command: c
re, im? -1 -1
Closest node is 0.00+0.00i
Command: c
re, im? 1 1
Closest node is 2.00+2.00i
Command: h
re, im? 10 10
Command: p
[10.00+10.00i,2.00+2.00i,0.00+0.00i]
Command: c
re, im? 8 8
Closest node is 10.00+10.00i
Command: c
re, im? 3 3
Closest node is 2.00+2.00i
Command: q
Bye!
```

Task 6. Implement the 'd' command to display the list of coordinates graphically in the complex plane (X axis is real, Y axis is complex). (You can skip this task initially and return to it later, if you wish.)
To display a list of complex numbers you must first compute the minimum and maximum of the real and imaginary parts (`minre`, `maxre`, `minim`, `maxre`) by traversing the list. Then you must compute the X and Y range (`rangere = maxre-minre`, `rangeim = maxim-minim`). You can use the following code snippets:

```
#define MAX(a,b) ((a)<(b)?(b):(a))
#define MIN(a,b) ((a)>(b)?(b):(a))
#define HEIGHT 50
#define WIDTH 50
int main (void) {
  pixel display[HEIGHT][WIDTH];
  init_display (HEIGHT, WIDTH, 10, display);
  ..
}
```

To display complex number `(r,i)` in the graphical window at `pixel[row][column]` consider the figures below.

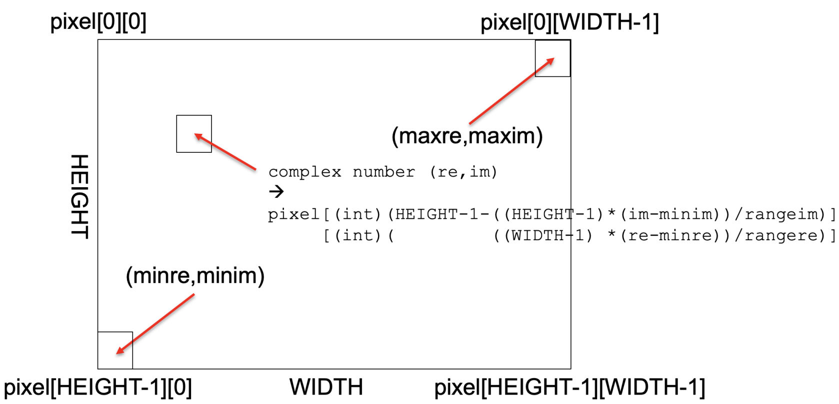

Figure 2: Display window and coordinate systems.

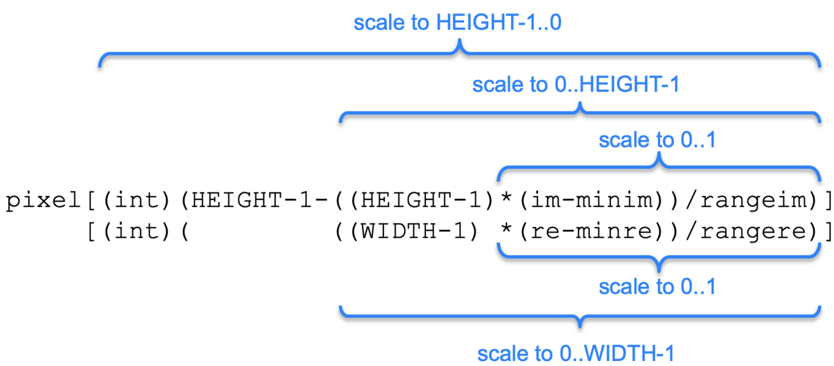

Figure 3: Converting a complex number to display window coordinates.

Hint: When the list is empty, just clear the display with `clear_display()`. You'll also notice that when there is only one complex number then the real and imaginary ranges (`rangere` and `rangeim`) are zero, leading to an error. The easiest way to solve this is to increase the ranges:

```
// range = 0 when we only have one point
if (minre == maxre) { minre--; maxre++; }
if (minim == maxim) { minim--; maxim++; }
```

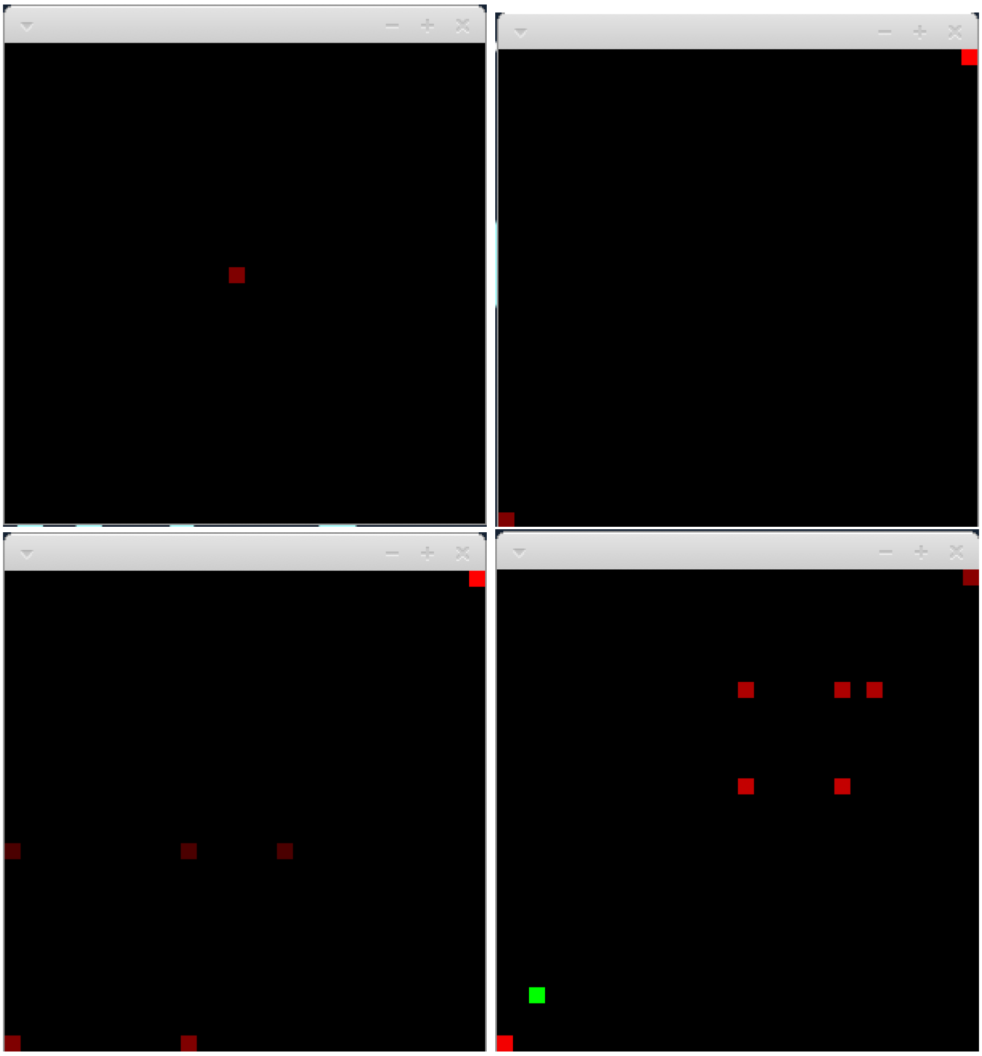

Figure 4: Example displays with one, two, and six complex numbers. The numbers are: -1-i, 8+7i, 7+5i, 5+7i, 5+5i, 10+10i. The last display also displays the 0+0i complex number in green (if it is in range). If you feel adventurous, you can scale the intensity of the complex number with its distance from 0+0i (intensity 255 for the point closest to 0 and intensity 128 for the point furthest from 0).

Task 7. Now let's insert new numbers in the middle of the list with the 'a' command that inserts a new complex number after the closest number in the list. The function `struct node_t *insert_after_closest (struct node_t *head, float re, float im)` first finds the closest complex number (using the function you wrote before), and then inserts the new number after it. If there are multiple closest numbers then insert after the first one.

```
Command: p
[]
Command: a
re, im? 0 0
Command: p
[0.00+0.00i]
Command: a
re, im? 1 1
Command: p
[0.00+0.00i,1.00+1.00i]
Command: a
re, im? 0.1 0.1
Command: p
[0.00+0.00i,0.10+0.10i,1.00+1.00i]
Command: a
re, im? 10 10
Command: p
[0.00+0.00i,0.10+0.10i,1.00+1.00i,10.00+10.00i]
Command: a
re, im? -10 -10
Command: p
[0.00+0.00i,-10.00-10.00i,0.10+0.10i,1.00+1.00i,10.00+10.00i]
Command: q
Bye!
```

Task 8. Now let's insert new numbers in the middle of the list with the 'b' command that inserts a new complex number *before* the closest number in the list. The function `void insert_before_closest (struct node_t **head, float re, float im)` first finds the closest complex number (using the function you wrote before), and then inserts the new number before it. If there are multiple closest numbers then insert before the first one. Note that while insert after used pass by value for the `head` parameter, now the `head` parameter is passed by reference and the function returns `void`. (To help you practice with different parameter-passing styles.)

```
Command: p
[]
Command: b
re, im? 0 0
Command: p
[0.00+0.00i]
Command: b
re, im? 1 1
Command: p
[1.00+1.00i,0.00+0.00i]
Command: b
re, im? 0.1 0.1
Command: p
[1.00+1.00i,0.10+0.10i,0.00+0.00i]
Command: b
re, im? 10 10
Command: p
[10.00+10.00i,1.00+1.00i,0.10+0.10i,0.00+0.00i]
Command: b
re, im? -10 -10
Command: p
[10.00+10.00i,1.00+1.00i,0.10+0.10i,-10.00-10.00i,0.00+0.00i]
Command: q
Bye!
```

Task 9. To practise recursion, write a recursive function `void print_list_reverse (struct node_t *head)` that prints the list in reverse order and implements the command 'v'.

```
Command: v
[]
Command: h
re, im? 0 0
Command: t
re, im? 1 1
Command: t
re, im? 2 2
Command: p
[0.00+0.00i,1.00+1.00i,2.00+2.00i]
Command: v
[2.00+2.00i,1.00+1.00i,0.00+0.00i]
Command: q
Bye!
```

Task 10. Now let's remove numbers from the list. First, write a function `struct node_t *remove_front (struct node_t *head)` that removes the first number from the front of the list (the 'f' command). The list is unchanged if it is already empty.

```
Command: p
[]
Command: f
Command: p
[]
Command: h
re, im? 1 1
Command: h
re, im? 2 2
Command: p
[2.00+2.00i,1.00+1.00i]
Command: f
Command: p
[1.00+1.00i]
Command: f
Command: p
[]
Command: q
Bye!
```

Task 11. Write a function `void remove_end (struct node_t **head)` that removes the last number at the end of the list (the 'e' command). The list is unchanged if it is already empty.

```
Command: p
[]
Command: e
Command: p
[]
Command: h
re, im? 1 1
Command: h
re, im? 2 2
Command: p
[2.00+2.00i,1.00+1.00i]
Command: e
Command: p
[2.00+2.00i]
Command: e
Command: p
[]
Command: q
Bye!
```

Task 12. Now let's remove a number in the middle of the list with the 'r' command that removes the closest complex number in the list. The function `struct node_t *remove_closest (struct node_t *head, float re, float im)` first finds the closest complex number (using the function you wrote before), and then removes it. If there are multiple closest numbers then remove the first one.

```
Command: h
re, im? 0 0
Command: t
re, im? 10 10
Command: a
re, im? 1 1
Command: p
[0.00+0.00i,1.00+1.00i,10.00+10.00i]
Command: b
re, im? 9 9
Command: p
[0.00+0.00i,1.00+1.00i,9.00+9.00i,10.00+10.00i]
Command: r
re, im? 2 2
Command: p
[0.00+0.00i,9.00+9.00i,10.00+10.00i]
Command: r
re, im? 9.5 9.5
Command: p
[0.00+0.00i,10.00+10.00i]
Command: q
Bye!
```

Submission: Your final solution must be submitted through OnCourse which will automatically grade this submission. Upload your C program to Oncourse (`Exercise 6: Lists (due 14 Oct 23:55)`). You can resubmit as often as you want until the deadline.

function	1	2	3	4	5	6	7	8	9	10	11	12	% per fn	cumulative %
quit	1	1	1	1	1	1	1	1	1	1	1	1	9%	9%
tail		1			1						1	1	5%	14%
print		1	1	1	1	1	1	1	1	1	1	1	5%	18%
head			1	1						1		1	9%	27%
closest				1								1	9%	36%
display					1							1	9%	45%
after						1			1			1	9%	55%
before							1	1				1	9%	64%
reverse								1				1	9%	73%
front									1			1	9%	82%
end										1		1	9%	91%
remove											1	1	9%	100%

Figure 5: Scoring of functions