Writing and Testing readInput

coursera.org/learn/interacting-system-managing-memory/supplement/SSKpc/writing-and-testing-readinput

Now we turn out attention to writing the functions **readInput**, **getClassList**, **writeAllFiles**, **freeClasses**, and **freeRoster**.

We will leave working through the details (Steps 1–4) of **readInput** to the reader, but one probably comes up with code that looks like this:

```
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
  if (f == NULL) {
```

```
    return NULL; //Could not open file->indicate failure
  }
  roster_t * answer = malloc(sizeof(*answer));
  answer->numStudents = readAnInteger(f);
  answer->students = malloc(answer->numStudents * sizeof(*answer-
>students));
  if (answer->students == NULL) {
    free(answer);
    return NULL;
  }
  for (int i =0; i < answer->numStudents; i++) {
    answer->students[i] = readAStudent(f);
  }
  //we could check if we have reached EOF here---
  //depends on what we consider "correct" for the format
  //Directions don't specify if this is an error.
  int result = fclose(f);
    assert (result == 0);
    return answer;
}
```

Note that again, we have formed two new programming tasks to do—writing **readAnInteger** and **readAStudent**. While this may seem counterproductive—we have gone from one programming task (solving this entire problem), to five programming tasks (as listed above), to six programming tasks (the remaining four from above plus the new two)—we have actually made significant progress. Each of our remaining tasks is far smaller and simpler than the ones that motivated their creation.

Furthermore, by writing functions for each logical task, we will reduce code duplication. We will find that the same task comes up in multiple contexts, and be able to just call a function we already wrote. For example, we will find that **readAnInteger** is useful in **readAStudent**, which might look like (again, we leave Steps 1–4 to the reader):

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
void stripNewline(char * str) {
  char * p = strchr(str, '\n');
  if (p != NULL) {
    *p = '\0';
  }
```

```
}


student_t * readAStudent(FILE * f) {
  student_t * s = malloc(sizeof(*s));
  size_t sz = 0;
  s->name = NULL;
  getline(&s->name, &sz, f);
  stripNewline(s->name);
  s->numClasses = readAnInteger(f);
  s->classes = malloc(s->numClasses * sizeof(*s->classes));
  for (int i = 0; i < s->numClasses; i++) {
    s->classes[i] = NULL;
    getline(&s->classes[i], &sz, f);
    stripNewline(s->classes[i]);
  }
  return s;
}
```

At this point, we have only introduced one new (rather simple) additional function,**stripNewline** (which just removes the '\n' from the end of the string), which we also show above. When we write the **readAnInteger** function (which we will not show here, but you should be able to write it), we will not need any other functions at all—the only complex steps in **readAnInteger** can be implemented by calling functions in the C library. At that point, we will have completed everything that we needed for **readInput**.

Contrast these four functions with what we wrote previously. Even though these four functions perform the same behavior as the giant function we saw earlier, they are much more readable, debuggable, and maintainable.

As you write each of these functions, you would want to test them. Testing each portion of the code in isolation gives us many benefits. You can debug any problems you have here now, and be confident that each works before you proceed (which involves making functions that rely on these smaller functions).

Testing the smaller functions is relatively straightforward (you will want to write test harnesses for them, withmain functions that are separate from the rest of your program). However, **readInput** is a bit larger and more complex—you will need a way to see if it got the right output easily to test it. To perform this testing, we could write another function that we do not really need for our main task **void printClassRoster(roster_t * r)**, which prints out the class roster. We then change main (or write a separate main in another file) to temporarily look like this:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
int main(int argc, char ** argv) {
  if (argc != 2) {
    fprintf(stderr, "Usage: roster inputname\n");
    return EXIT_FAILURE;
  }
```

```
  roster_t * the_roster = readInput(argv[1]);
  if (r == NULL) {
    fprintf(stderr,"Could not read the roster information\n");
    return EXIT_FAILURE;
  }
  printClassRoster(the_roster);  //print what we read.
  //skip the rest: it isnt written yet anyways
  /* classes_t * unique_class_list = getClassList(the_roster); */
  /* writeAllFiles(unique_class_list, the_roster); */
  /* freeClasses(unique_class_list); */
  /* freeRoster(the_roster); */
  return EXIT_SUCCESS;
}
```

Completed