Finishing the Program

We would then proceed with **getClassList** in much the same fashion—working Steps 1–5, and abstracting complex steps out into their own functions. In this case, we will likely want to abstract out a function to check if an array of strings contains a particular string (this function will also prove useful when we go to write our output files), and may or may not want to abstract out the function to add a string to the list of classes (which requires reallocing the array). We might end up with a function like this:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
classes_t * getClassList(roster_t * the_roster) {
  classes_t * ans = malloc(sizeof(*ans));
  assert(ans != NULL);
  ans->numClasses = 0;
  ans->classNames = NULL;
```

```
  for (int i = 0; i < the_roster->numStudents; i++) {
    student_t * current_student = the_roster->students[i];
    for (int j = 0; j < current_student->numClasses; j++) {
      if (!contains(ans->classNames, current_student->classes[j],
          ans->numClasses)) {
        addClassToList(ans, current_student->classes[j]);
      }
    }
  }
  return ans;
}
```

The two pieces we abstracted out are **contains**:

```
1
2
3
4
5
6
7
8
int contains(char ** array, const char * str, int n) {
  for (int i = 0; i < n; i++) {
    if (!strcmp(array[i],str)) {
      return 1;
    }
  }
```

```
  return 0;
}
```

and **addClassToList**:

```
1
2
3
4
5
6
7
void addClassToList(classes_t * class_list, char * name) {
  class_list->numClasses++;
  class_list->classNames = realloc(class_list->classNames,
                                   class_list->numClasses *
                                      sizeof(*class_list->classNames));
  class_list->classNames[class_list->numClasses - 1] = name;
}
```

After writing these functions, we should again test everything together we have so far by writing a function to print the list of classes, and putting a call to it in main.

Our last programming task is to write our output files. We might end up with something like this:

```
1
2
3
4
```

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

```
void writeOneFile(const char * cName, roster_t * r) {
  char * fname = makeRosterFileName(cName);
  FILE * output_file = fopen(fname, "w");
  if (output_file == NULL) {
    perror("fopen");
    fprintf(stderr,"Trying to open %s\n", fname);
    abort();
  }
```

```
  free(fname);
  for (int i = 0; i < r->numStudents; i++) {
    student_t * s = r->students[i];
    if (contains (s->classes, cName, s->numClasses)) {
      fprintf (output_file, "%s\n", s->name);
    }
  }
  int result = fclose(output_file);
  assert(result == 0);
}

void writeAllFiles(classes_t * unique_class_list, roster_t * the_roster) {
  for (int i = 0; i < unique_class_list->numClasses; i++) {
    writeOneFile(unique_class_list->classNames[i], the_roster);
  }
}
```

This code abstracts out the task of writing one file into its own function, leaving the **writeAllFiles** to simply iterate over the class list, and let **writeOneFile** do the major work. The **writeOneFile** function abstracts out **makeRosterFileName**, as that was itself a somewhat complex step in translating the generalized algorithm into code. We can then write this function:

```
1
2
3
4
5
6
7
```

```
char * makeRosterFileName(const char * cName) {
  const char * suffix = ".roster.txt";
  unsigned len = strlen(cName) + strlen(suffix) + 1;
  char * ans = malloc(len * sizeof(*ans));
  snprintf(ans, len, "%s%s", cName, suffix);
  return ans;
}
```

At this point, we would have completed the programming task. We wrote 14 functions, but each function is a reasonable size (all are less than 25 lines) with a clearly defined purpose. Notice how this abstraction contributes to the readability of the code. You can look at a function, and it is small enough that you can think about what it does. You can then understand the higher-level functions in terms of the lower-level functions they call—knowing *what* they do, without having to worry about *how* they do it at that point. Such is the benefit of good abstraction.

Additionally, our code is easy to change. For example, suppose we decide to change the format of the input file so that each student's entry does not start with the number of classes that they have; but instead, their entry ends with a blank line. We would only need to change **readAStudent**, as that code is the only part that handles reading a student's information. Everything else is isolated from the details of *how* to read a student by abstracting that task into the**readAStudent** function. The rest of the code (in particular, **readInput**) only needs to know that**readAStudent** will read one student entry and return a pointer to it. If we made more drastic changes to the input format, then the impact on our code changes would be limited to **readInput** and the functions it calls. Nothing else cares about the details of *how* the input is read.