Planning the High-Level Algorithm

As always, we should begin with Step 1, and work through the programming process we have learned all throughout this book. Of course, for a task this big, we should recognize that we are seeking to break the task down into many functions, so we should be content with relatively complex steps—which then dictate what functions we need.

We can make a couple important steps towards solving this problem. The first is that we can define several of the types involved in the problem. The video worked through how we see our **student_t**and **roster_t**types. We can similarly work through what we need to represent our list of all the classes for a **classes_t** type. We would then come up with the following declarations:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
20

21

// Each student has a name and an array of classes

struct student_tag {

  char * name;

  char ** classes;

  int numClasses;

};

typedef struct student_tag student_t;


//A roster is an array of student records (with the size)

struct roster_tag {

  student_t ** students;

  int numStudents;

};

typedef struct roster_tag roster_t;


//An array of class names (with the size)

struct classes_tag {

  char ** classNames;

  int numClasses;

};

typedef struct classes_tag classes_t;
```

Having these type declarations in mind will be helpful as we work through the sub-problems that we are about to create. These types will be the input and/or output types of many of these other algorithms. Knowing exactly what they look like allows us to draw precise pictures in Step 1 for each of those problems, coming up with the right algorithms.

The other important step we took in the video was coming up with the general algorithm for **main**:

```
1
2
3
Read the input from the file named by argv[1] (call this r)
Make a list of all of the (unique) class names (call this result c)
Write one output file per class (from c and r)
```

We will want to make a few alterations to this algorithm. First, we will notice that we did not write down anything about returning our answer—however, main always returns an int. In this case, we should return EXIT_SUCCESS. As we think carefully about what our other functions (that we are about to write) will do, we may realize that they will need to **malloc** some memory—so we should **free** that memory when we are done with it. If we do not realize this fact yet, we may need to come back later and add the freeing. Furthermore, if we test carefully (with corner cases) in Step 4, we might realize we need to add some error checking. Ultimately, this results in a nice generalized algorithm for main:

```
1
2
3
4
5
6
7
8
Check that the argc == 2 (fail if not)
Read the input from the file named by argv[1] (call this the_roster)
Check that r != NULL (fail if not)
Create the list of all of the class names (call this result unique_class_list)
Write all the class roster files from unique_class_list and the_roster
Free memory held by unique_class_list
```

```
Free memory held by the_roster

return EXIT_SUCCESS
```

As long as we abstract each of the complex steps out into its own function, this results in a nice, short, easy to understand main function:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
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 (the_roster == NULL) {
```

```
    fprintf(stderr,"Could not read the roster information\n");
    return EXIT_FAILURE;
  }
  classes_t * unique_class_list = getClassList(the_roster);
  writeAllFiles(unique_class_list, the_roster);
  freeClasses(unique_class_list);
  freeRoster(the_roster);
  return EXIT_SUCCESS;
}
```

Having written main, we now have five new programming tasks, which will each be a function: **readInput**, **getClassList**, **writeAllFiles**, **freeClasses**, and **freeRoster**.