Commenting and Documentation

Well-documented code is significantly easier to read than poorly-documented code. Good documentation provides insights into the algorithm being used, explains why things that may seem surprising are correct, and generally allows the reader to understand how and why the code works the way it does. Of course, the key here is in writing *good* documentation—not just writing a lot of documentation.

Most documentation is written as comments in the code. We have seen comments before and already learned that they indicate text that is for humans, and thus is ignored by the compiler. In C and C++, comments can either be written with //to comment to the end of the line or /* */to comment everything enclosed by the slash-star and star-slash.

```
3

   is a comment */
```

Understanding how to write good documentation can be quite tricky for novice (and even moderately experienced) programmers. Skill in this area generally increases as you read other people's code (or your own code that you have not seen in a long time) and find comments particularly useful, or lacking—things you wish the comments explained. Here are some good rules of thumb to help write better documentation:

Document large-scale design

As you design larger programs, you will have multiple pieces that fit together. Describe how they fit together. What are the interfaces between modules? What are the invariants of the whole system?

Describe Each Component

For each component (function, object (if you get to C++), or file) of your system, write a comment at the start that describes it. Start by describing the interface, for anyone who just needs to use the component and does not need to know about its implementation details. For functions, describe what each parameter means, as well as any restrictions on their values. Describe what the function returns. If it has any side effects, explain them.

Continue by describing implementation details for anyone who needs to modify it. If it uses a commonly known algorithm, say so. If it uses an algorithm you designed yourself, explain how the algorithm works.

Do Not Comment The Obvious

A common mistake is to believe that just writing a lot of comments makes for good documentation. However, comments that describe the obvious are counterproductive—they clutter up the code while providing no useful information. Consider the following example:

```
1
2
3
4
5
6
7
int sumNums (int n) {
  int total = 0;  //declare total, set to 0
  for (int i = 0; i < n; i++) { //for loop
    total = total + n; //add n to the total
  }
  return total; //return the total
}
```

Here, the programmer has written four comments, but none of them contribute anything useful to understanding the code. Anyone who understand the basics of how C works can get the same information instantly just by looking at the code.

A better comment at the start of the function gives more information:

```
1
2
3
4
5
```

```
6

7

8

9

10

/* adds numbers from 0 to n (exclusive) and

 * returns total

 */

int sumNums (int n) {

  int total = 0;

  for (int i = 0; i < n; i++) {

    total = total + n;

  }

  return total;

}
```

Explain The Unexpected

If your code contains something unusual, such as a statement that is correct but could appear to be erroneous, you should document why that statement is correct. When someone unfamiliar with the code reads it, this documentation will save them from wondering whether or not the unusual aspect of the code is an error—they can simply read the explanation, rather than trying to figure out if it is broken. In general, if you think someone would ask "Why does it do that?" or "Is that right?" you should preemptively answer that question with a comment.

Consider the following example of code with useful documentation in it:

```
35
36
37
38
39
40
32
33
34
29
30
31
24
25
26
27
28
19
20
21
22
23
16
17
18
14
15
```

```
11
12
13
3
4
5
6
7
8
9
10
1
2
    array[j] = temp;
  }
  //swap array[i] with the pivot (array[n-1])
  array[n-1] = array[i];
  array[i] = pivot;
  //recursively sort the partitioned halves: [0,i) and [i+1,n)
    //swap array[i] with array[j]
    int temp = array[i];
    array[i] = array[j];
    if (i >= j) {  //if i and j have crossed, data is partitioned.
      break;
    }
      i = i+1;
    } while (array[i] < pivot);
    do {  //scan from right for value < pivot
```

```
      j = j-1;
    } while (j >= 0 && array[j] >= pivot );
  //we just use array[n-1] here for simplicity. In a real implementation,
  //we might want more sophisticated pivot selection (see CLR)
  int pivot = array[n-1];
  while (1) {
    do { //scan from left for value >= pivot
  //j starts at n-1 even though it is decremented before it
  //is used to index the array, since array[n-1] is our pivot.
  int j = n-1;
  //i starts at -1: it is incremented before it is used to index the array
  int i = -1;
  if (n <= 1) {
    return; //arrays of size 1 or smaller are trivially sorted
  }
 * n (the number of items in 'array')
 * and sorts the integers found in 'array' into ascending order.
 * The sorting algorithm used here is the widely-known "quick sort."
 * For details, see
 * - "All of Programming" (Hilton + Bracy), Chapter 26, or
 * - "Introduction to Algorithms" (CLRS), Chapter 7.
 */
void sort(int * array, size_t n) {
/* This function takes two parameters:
 * array (which is an array of ints)
```

The code begins with a description of the function from a high level. It describes the interface to the function; if you only need to use the function, you can tell how to do so from just reading the comment. It also describes the algorithm used in this function. In this particular case, since the algorithm is well known, it provides a brief description and a couple of references in case the reader is unfamiliar with it. Note that this function is a bit long (it does not quite fit into my terminal), so ideally, we should abstract part of it out into its own separate function. The part in lines 14–36 is a great candidate for pulling out into its own function, as it performs a specific logical task, called "partitioning" the data.

Inside the code, the comments describe why or how things happen, as well as the rationale behind things that seem unusual. For example, the comment describing the initialization of **i**explains why it is initialized to -1, which may be confusing (or even seem like a mistake) to the unfamiliar reader.

Completed