

Designing the Algorithm for the Program

After reading the program specification, it should be immediately clear that there are three basic steps to be performed:

- I. Read the input data.
- II. Perform the specified calculations.
- III. Write the specified results.

Considering Step I, we must be guided by the specified format of the input file:

Origin	Destination	Miles	Time
-----	-----	-----	-----
Pecos, TX	Carslbad, NM	75	1:07

The first two lines are irrelevant as far as the program is concerned, so they must be read and discarded, or somehow skipped over. The third line is the interesting part. Logically we have to read four "tokens": the name of the origin, the name of the destination, the length of the trip in miles, and the time required. So we might expand Step I as:

- I. Read the input data.
 - A. Read the name of the origin.
 - B. Read the name of the destination.
 - C. Read the mileage.
 - D. Read the time.

However, the time is a composite value (i.e., it's made up of separate parts). We must read the hours and minutes separately, probably to combine later into a total time value. So we have a revision:

- I. Read the input data.
 - A. Read the name of the origin.
 - B. Read the name of the destination.
 - C. Read the mileage.
 - D. Read the time.
 - i. Read the hours.
 - ii. Get rid of the colon.
 - iii. Read the minutes.

Apparently, each origin and destination name will consist of a place name, followed by a comma, followed by a state abbreviation. But, there's nothing in the specification that requires us to handle those separately, and there is nothing in the specification that guarantees the names will always have that format. So we will treat each name as a unit and read it all at once.

Before going on to Step II we should make some provisions for writing things down more precisely. The values we will be reading must be stored in variables, and the names of those variables will be useful in describing the rest of our algorithm. So we'll add variable names to the expanded design for Step I:

- I. Read the input data.
 - A. Read the name of the origin into `tripOrigin`.
 - B. Read the name of the destination into `tripDestination`.
 - C. Read the mileage into `tripMiles`.
 - D. Read the time.
 - i. Read the hours into `tripHours`.
 - ii. Get rid of the colon.
 - iii. Read the minutes into `tripMinutes`.

I use the convention that variable names are written in a different font (Courier New here) to make them stand out more clearly in the outline.

Now consider Step II. The two values to be calculated are described clearly in the specification, so that's not a problem.

- II. Perform the specified calculations.
 - A. Calculate the total time in minutes.
 - B. Calculate the average speed in MPH.

However, the specification doesn't give formulas for calculating those values, so we must figure them out... might as well do that now as put it off. Neither one is particularly difficult. In Step I, we've read values for `tripHours` and `tripMinutes`. The total time, in minutes would just be:

```
tripTime = 60 * tripHours + tripMinutes.
```

(If that's not clear, consider a particular example. How would you convert 3:30 into minutes? That's 3 hours plus 30 minutes. Now one hour is 60 minutes, so 3 hours would be $3 * 60$ or 180 minutes. So 3:30 would be $180 + 30 = 210$ minutes.)

For the average speed, recall that: $\text{distance} = \text{rate} * \text{time}$. A little algebra: $\text{rate} = \text{distance} / \text{time}$. So, in our case, we could say:

```
tripMPH = tripMiles / tripTime.
```

But that's not quite right... considering the units on the right-hand side. We have miles divided by minutes, so this would give us the average speed in miles per minute, not miles per hour. For this calculation, we need miles divided by hours, so we must convert the time into hours. That's easy enough; just divide the total minutes by the number of minutes in an hour:

```
tripMPH = tripMiles / (tripTime / 60).
```

It's slightly more attractive as:

```
tripMPH = 60 * tripMiles / tripTime.
```

So we expand Step II as:

- II. Perform the specified calculations.
 - A. Calculate `60 * tripHours + tripMinutes` and store the result in `tripTime`.
 - B. Calculate `60 * tripMiles / tripTime` and store the result in `tripMPH`.

That seems to cover Step II. Now consider Step III. Again, the specification is clear about what has to be written out. It helps to look at the sample output (if someone was kind enough to provide any). In any case, you must read the output specification carefully. This is pretty straightforward:

- III. Write the output data.
 - A. Write the identification information.
 - i. Write "Programmer: " and your name.
 - ii. Write the assignment title, "CS 1044 Notes Example ", on the next line.
 - iii. Write a blank line.
 - B. Write the table header.
 - i. Write the specified column labels on the next line.
 - ii. Write the top table boundary on the next line.
 - C. Write the table body.
 - i. Write `tripOrigin` and `tripDestination`, nicely formatted, on the next line.
 - ii. Write `tripMiles` and `tripTime`, nicely formatted.
 - iii. Write `tripMPH`, nicely formatted, with one digit after the decimal point.
 - D. Write the bottom table boundary on the next line.

So we have the following complete design outline:

- I. Read the input data.
 - A. Read the name of the origin into `tripOrigin`.
 - B. Read the name of the destination into `tripDestination`.
 - C. Read the mileage into `tripMiles`.
 - D. Read the time.
 - i. Read the hours into `tripHours`.
 - ii. Get rid of the colon.
 - iii. Read the minutes into `tripMinutes`.
- II. Perform the specified calculations.
 - A. Calculate $60 * \text{tripHours} + \text{tripMinutes}$ and store the result in `tripTime`.
 - B. Calculate $60 * \text{tripMiles} / \text{tripTime}$ and store the result in `tripMPH`.
- III. Write the output data.
 - A. Write the identification information.
 - i. Write "Programmer: " and your name.
 - ii. Write the assignment title, "CS 1044 Notes Example ", on the next line.
 - iii. Write a blank line.
 - B. Write the table header.
 - i. Write the specified column labels on the next line.
 - ii. Write the top table boundary on the next line.
 - C. Write the table body.
 - i. Write `tripOrigin` and `tripDestination`, nicely formatted, on the next line.
 - ii. Write `tripMiles` and `tripTime`, nicely formatted.
 - iii. Write `tripMPH`, nicely formatted, with one digit after the decimal point.
 - D. Write the bottom table boundary on the next line.

The design doesn't take into account anything about the language we'll be using to implement it. That's generally the way it should be, at least for simple programs. There are many details still to be handled. What kinds of variables will we use to store the data? Exactly how will we read the input data and write the output? Those decisions will usually depend on the particular language we will use to express the algorithm.

It's implicit in the process so far that we've been checking the design by imagining actually carrying out the steps and seeing if they do, in fact, produce the results we want. At this point you should do that with the entire algorithm outline. Start with the sample input given in the program specification, and work out the steps of the outline with paper and pencil. You should get the correct results, as given in the program specification. If you don't, then something's wrong.

The next step in the development process is to translate the algorithm design into C++. In the following discussion I assume that you've covered the material in the first four chapters of the CS 1044 Notes, through I/O streams and formatting output.

The design outline isn't just external documentation of your analysis of the problem and how to solve it. The outline can also serve as a framework when you develop the implementation in C++. We'll begin by formatting the outline as a comment; then we'll add the necessary C++ code to complete the implementation of each step in the outline. So, in the end, the design outline will become part of the internal documentation of the program itself.

Implementing the Program

We begin with the input code and the usual framework for a C++ program. We know that the variables we've identified so far must be declared, so we'll add those declarations immediately. We also know we need an input stream connected to the input file, so we'll take care of that as well. The variable types are straightforward. The origin and destination are strings of characters, so we use `string` variables to store them. The mileage and time values are integers, so we use `int` variables to store those.

```
// Includes section:
#include <iostream> // for cout
#include <fstream> // for file streams
#include <string> // for string variables
#include <climits> // for INT_MAX
using namespace std; // to put all of the above things in scope

int main() {

    ifstream In("TripData.txt"); // Attach an input stream to the input
                                // file.

    // Variables to store the input data:
    string tripOrigin, // Name of starting point for trip.
           tripDestination; // Name of destination for trip.
    int tripMiles; // Length of trip in miles.
    int tripHours, // Time trip took is given as hh:mm so:
        tripMinutes; // hours field
                    // minutes field

    // I. Read the input data.
    // A. Read the name of the origin into tripOrigin.
    In.ignore(INT_MAX, '\n'); // Skip over the two header lines in the
    In.ignore(INT_MAX, '\n'); // input file.
    getline(In, tripOrigin, '\t');

    // B. Read the name of the destination into tripDestination.
    getline(In, tripDestination, '\t');

    // C. Read the mileage into tripMiles.
    In >> tripMiles;

    // D. Read the time.
    // i. Read the hours into tripHours.
    // ii. Get rid of the colon.
    // iii. Read the minutes into tripMinutes.
    In >> tripHours;
    In.ignore(1, ':');
    In >> tripMinutes;
    In.close(); // close the input file

    . . .
}
```

One thing came up here that wasn't explicitly covered in the design outline. Reading the trip origin requires some preparation since there's data preceding it in the input file. So we added C++ code to get rid of that data.

We also had to decide what C++ features to use to read each input value. Since the origin and destination are just strings, and are terminated by tab characters, we use the `getline()` function to read those and store them in the proper variables. The remaining data is numeric, so we use the extract operator to read it. The colon could be read into a char variable, using `extract` or `get()`, but there's no reason to save it so we just use `ignore()` to get rid of it.

We should test the operation of this code before moving on to the remainder of the implementation. After all, if we aren't reading the input data correctly then we can't do any useful calculations yet. We can test the code above by adding simple output code to print the values of the variables we've read data into:

```
cout << tripOrigin << endl;
cout << tripDestination << endl;
cout << tripMiles << endl;
cout << tripHours << "    " << tripMinutes << endl;
```

If this doesn't echo the values given in the input file, we've done something wrong. Once we've verified that the input code is working properly we will delete this code since it's not needed in the final version.

That takes care of the implementation of the input code. We are ready to add the calculations. In this case the calculations are relatively trivial, and we've done the hard part already (determining the variables and formulas).

We do have a literal constant, 60, which is used in our time conversion. We'll make that a named constant in the implementation. The trip time is an integer, and the trip speed is a decimal value, so the corresponding variables should be of types `int` and `double` respectively. Finally, we have the typical integer division problem when we calculate the speed. The distance and the time are both stored as integers. If we simply divide them the result will be truncated to an integer, which is not what we want. So, we'll explicitly convert them to type `double`. (It would suffice to convert one explicitly, but it's clearer and no less efficient to convert both explicitly.)

So we add the following:

```
. . .
const int MINPERHOUR = 60;

// Variables to store calculated results:
int    tripTime;           // Time for trip in minutes.
double tripMPH;           // Speed of trip, in miles per hour.

// II. Perform the specified calculations.
//    A. Calculate 60 * tripHours + tripMinutes and store the
//        result in tripTime.
tripTime = MINPERHOUR * tripHours + tripMinutes;

//    B. Calculate 60 * tripMiles / tripTime and store the
//        result in tripMPH.
tripMPH = MINPERHOUR * double(tripMiles) / double(tripTime);
. . .
```

Again we have a testing issue. We should add code to print these calculated values in order to verify that the code is (likely to be) correct. The following code will do:

```
cout << tripTime << endl;
cout << setprecision(1) << tripMPH << endl;
```

If these produce the expected results then we may have some faith in the calculation code. If not, then we must correct the calculations.

Now on to the output code. We'll need an output stream, connected to the appropriate file. The rest follows directly from the project specification and is reasonably straightforward. The main issue is how to format the output neatly with respect to the column headers. The simplest approach is to use `setw()` to set field widths to line things up, and change the justification setting to position the strings and numbers as we want.

```

. . .
ofstream Out("Summary.txt"); // Attach an output stream to the
                             // output file.
Out << fixed << showpoint;    // Prepare output stream for decimal
                             // output.

// III. Write the output data.
//     A. Write the identification information.
//         i. Write "Programmer: " and your name.
//         ii. Write the assignment title, "CS 1044 Notes Example ",
//             on the next line.
//         iii. Write a blank line.

Out << "Programmer: " << "Bill McQuain" << endl;
Out << "CS 1044 Notes Example" << endl;
Out << endl;

//     B. Write the table header.
//         i. Write the specified column labels on the next line.
//         ii. Write the top table boundary on the next line.
const string HEADERS    = "Origin          Destination      "
                          "Mileage  Minutes    MPH";
const string DELIMITER  = "-----"
                          "-----";

Out << HEADERS << endl;
Out << DELIMITER << endl;

//     C. Write the table body.
//         i. Write tripOrigin and tripDestination, nicely
//            formatted, on the next line.
//         ii. Write tripMiles and tripTime, nicely formatted.
//         iii. Write tripMPH, nicely formatted, with one digit after
//             the decimal point.
Out << left; // left-justify names
Out << setw(20) << tripOrigin; // in 20-column fields
Out << setw(20) << tripDestination;
Out << right; // right-justify numbers
Out << setw(7) << tripMiles // in varied-width fields
    << setw(10) << tripTime
    << setw(8) << setprecision(1) << tripMPH // MPH with 1 decimal place
    << endl;

//     D. Write the bottom table boundary on the next line.
Out << DELIMITER << endl;

Out.close(); // close the output file
return 0;    // terminate the program
}

```

Of course, now we must execute the program on a variety of input files and verify that it produces correct output for those cases. Remember well that no amount of testing can prove that the implementation is entirely correct. However, a reasonable amount of testing will usually find most, if not all, of the flaws in a small program.