

Sentinel-controlled Input, Selection Mechanisms

This programming assignment uses many of the ideas presented in sections 3 through 6 of the course notes, so you are advised to read those carefully. Read and follow the following program specification carefully. This program is somewhat more complex than Project 2, so don't underestimate it.

This program will be worth 8% of your final grade and will be divided up into two parts – 80% will be for correct output (EAGS score) and 20% for design and style as described by the Programming Standards discussed later.

The Program Specification:

Party Music

You are putting together a music CD for a party and have arranged a list of songs in the order in which you want to play them. However, you do not know whether all the songs will fit on a single CD, which will hold 74 minutes of audio. You would like to fit as many songs as possible onto the CD, keeping them in the order you've already determined. This may require leaving out one or more of the songs you've selected.

Input file description and sample:

Your program **must** read its input from a file named `music.data` — use of another input file name will result in a score of zero. The first line of the input file contains a header, which should be ignored. The next several lines of the input file will each contain:

- A positive integer indicating the position of the song in the list. These song numbers will be in sequential order.
- A string indicating the type of music: Blues, Country, Hip Hop, Pop, or Rock.
- A nonnegative integer giving the minutes portion of the song length
- A nonnegative integer giving the seconds portion of the song length

The values will be separated by tabs, and a single newline character will follow the last value.

You may assume that all the input values will be logically correct (no negative or missing values, for instance). Here is an example:

```
Mix CD Input
1    Pop    8    33
2    Pop    8    32
3    Rock   3    25
4    Hip Hop 8    39
5    Country 8    19
6    Country 4    39
7    Hip Hop 7    12
8    Country 4    55
9    Pop    5    11
10   Hip Hop 5    49
11   Hip Hop 6    4
12   Pop    4    12
13   Pop    2    53
14   Blues  5    19
15   Rock   2    9
```

Note that you must **also not make any assumptions** about the number of lines of data in the input file. Your program must be written so that it will detect when there are no more lines in the input file.

What to Calculate:

Your program must determine which songs are to be included on the tape. When you read the data for a song, you will add that song to the tape if it will fit in the remaining space (time) available. If the song won't fit, you will reject it and go on to the data for the next song, if there is one.

When you add a song to the tape, you must also calculate the total time used, and the time remaining on the tape. For each type of song, you must count the total running time.

Output description and sample:

Your program must write its output data to a file named `tape.list` — use of any other output file name will result in a score of zero. A sample output file produced from the sample input file above is shown below:

```

Programmer: Chris Knestrick
Party Music

Song          Song Time          Total Time
Number      Minutes  Seconds  Minutes  Seconds
-----
  1           8         33         8         33
  2           8         32        17          5
  3           3         25        20         30
  4           8         39        29          9
  5           8         19        37         28
  6           4         39        42          7
  7           7         12        49         19
  8           4         55        54         14
  9           5         11        59         25
 10           5         49        65         14
 11           6          4        71         18
 15           2          9        73         27
-----

Time Remaining      Minutes  Seconds
Blues                0         0
Country             17         53
Hip Hop             27         44
Pop                 22         16
Rock                 5         34
    
```

The first line of your output should identify you by name, as shown. The second line should include the title "Party Music" only. The third line should be blank.

Next your output file will contain a table, with one line of output for each song given in the input file that fit onto the tape. Each line of the table should list the song name, the length of the song (in minutes and seconds), and the total time used on the tape. If the song will not fit on the tape, there should be no entry for it in the table. The table columns should have labels, exactly as shown. There should be a line of delimiters immediately after the column labels, and another to mark the end of the table. A newline character should follow each line of output (including the last).

After the table, your output file will display the amount of time left on the tape (in minutes and seconds) as well as the total running time of songs that are in each of the given categories. Again, you must use the format and labels shown below. The categories must also be shown in alphabetical order.

You are not required to use the exact horizontal spacing shown in the example above, but your output must satisfy the following requirements:

- You must use the specified header and column labels, and include your name in the first line as shown.
- You must arrange your output in neatly aligned columns, with a label identifying the contents of each column. Use spaces (not tabs) and `setw()` to align your output. Note that while the EAGS doesn't deduct points for horizontal alignment, the GTA who grades your source code for programming standards may do so.
- You must use the same ordering of the columns as shown here.
- You must print a newline at the end of each line.

Programming Standards:

For this program, you will be expected to submit a design outline in the header (worth half of your style grade). The design should be sufficiently detailed – at the lowest level, each entry should equate to 2 or 3 lines of actual code. Program designs are covered in Chapter 2 of the course notes (specifically, slides 2.3 and 2.4). When writing a program, the design comes first. Consequently, when you begin writing your program, you should already have a completed design in writing (though the design may change as the program is written). If you come to the GTA/instructor for help with the actual program (that is, you have started writing code), you will be **required** to produce your design document before any assistance is provided.

You'll be expected to observe good programming/documentation standards. All the discussions in class about formatting, structure, and commenting your code will be enforced. A copy of *Elements of Programming Style* is included with the course notes — if you don't have a copy I strongly suggest you read the on-line edition (available from the course web page). Some specifics:

- You must include header comments specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose.
- You must use indentation and blank lines to make control structures like loops and if-else statements more readable.
- Use named constants instead of variables where appropriate.
- Use nested if's rather than disjoint if's where appropriate.

Your submission that receives the highest score may be graded for adherence to these requirements, whether it is your last submission or not. If two or more of your submissions are tied for highest, the earliest of those will be graded. Therefore: implement and comment your C++ source code with these requirements in mind from the beginning rather than planning to clean up and add comments later.

Incremental Development:

You'll find that it's easier and faster to produce a working program by practicing incremental development. In other words, don't try to solve the entire problem at once. First, develop your design. When the time comes to implement your design, do it piece by piece. Here's a suggested implementation strategy for this project:

- First, write the code necessary to read the entire input file. To test your work, include code to write what you're reading (and nothing else) to the output file. There's not much point in worrying about processing the data further until you know you're reading it correctly, and stopping at the right point. If you have a problem later on in the program (computation or output), the GTAs have the right to ask you to demonstrate that this works correctly before providing any help in those areas.

- Second, add the code to decide whether to include the current song on the tape, and keep track of how much time has been used and how much time is left.
- Third, add the code to count the total running time of included songs in each category and print that information.
- Fourth, clean up your output format and be sure it matches the specification above.

Now you have a substantially complete program. At this point, you should clean up your code, eliminating any unnecessary instructions and fine-tuning the documentation you already wrote. Check your implementation and output again to be sure that you've followed all the specifications given for this project, especially those in the Programming Standards section above. At this point, you're ready to test your program on all the posted input/output examples before submitting your solution to the EAGS.

Hints:

This program requires that you know how to read until input failure. This can be found in the course notes on slides 4.22 – 4.25.

Trying to do math with minutes and seconds can be tricky – is there a way to combine the two?

Testing:

Obviously, you should be certain that your program produces the output given above when you use the given input file. However, verifying that your program produces correct results on a single test case does not constitute a satisfactory testing regimen.

At minimum, you should test your program on **all** the posted input/output examples given along with this specification. The same program that will be used to test your solution generated those input/output examples. You could make up and try additional input files as well; of course, you'll have to determine by hand what the correct output would be.

Pledge:

Each of your submissions to the EAGS must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
// On my honor:  
//  
// - I have not discussed the C++ language code in my program with  
// anyone other than my instructor or the teaching assistants  
// assigned to this course.  
//  
// - I have not used C++ language code obtained from another student,  
// or any other unauthorized source, either modified or unmodified.  
//  
// - If any C++ language code or documentation used in my program  
// was obtained from another source, such as a text book or course  
// notes, that has been clearly noted with a proper citation in  
// the comments of my program.  
//  
// - I have not designed this program in such a way as to defeat or  
// interfere with the normal operation of the Automated Grader.
```

Failure to include this pledge in a submission is a violation of the Honor Code.