School of Computing and Information Systems

comp20005 Engineering Computation
Semester 1, 2021
Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of loops, `if` statements, functions, and arrays, by writing a program that first reads some numeric data, and then performs a range of processing tasks on it. The sample solution that will be provided to you after the assignment is completed will also make use of structures (covered in Chapter 8), and you may do likewise if you wish. But there is no requirement for you to make use of `struct` types, and they will not have been covered in lectures before the due date.

Sequential Data

Scientific and engineering datasets are often stored in text files using *comma separated values* (`.csv` format) or *tab separated values* (`.txt` or `.tsv` format), usually with a header line describing the contents of the columns. The simplest framework for processing such data is to first read the complete set of input rows into arrays, one array per column of data, and then pass those arrays (and a single buddy variable) into functions that carry out the required data transformations and analysis. Your task in this project is to use that processing approach to analyze a set of delivery requirements, and determine how many delivery agents are needed. In the example described next the agents are "drones" and the delivery items are "meals", but there are many other scenarios where similar processing logic applies.

Suppose that a large kitchen receives phone and internet orders for meals. Once an order has been placed the meal is prepared by a chef and cooked. The preparation and cooking time can be accurately estimated based on past records, and from them each meal's RFD ("ready for delivery") time is computed, together with the relative coordinates of the delivery address, and passed to the "deliver planner". For example, the data file `meals0.tsv` contains these values in tab-separated format:

```
order   rfdhh   rfdmm   delivx  delivy
10443   19      01      1.83    2.15
28132   19      07      -3.45   1.18
34332   19      21      2.65    0.53
22098   19      26      -1.53   0.44
55432   19      38      3.10    0.89
```

There will always be a single header line in all input files, and then rows each containing five values separated by "tab" characters (`'\t'` in C). Once the first line has been bypassed (write a function that reads and discards characters until it has read and discarded a newline character, `'\n'`), each data line can be read as a set of `int` and `double` variables using `scanf("%d%d%d%lf%lf",...)`, with five values per row. The five values in each row represent:

- `order`: an order number;
- `rfdhh`: the hour number within the day that the order will be ready for delivery;
- `rfdmm`: the minute number within the day that the order will be ready for delivery;
- `delivx`: the x-coordinate of the delivery address, in kilometers east/west of the kitchen; and
- `delivy`: the y-coordinate of the delivery address, in kilometers north/south of the kitchen.

Each data file contains the orders from within a single twenty-four hour period from 00:00 to 23:59 in a single day. This small example shows five meals that need to be delivered in the period between 7pm and 8pm one evening. In general, data files might contain dozens or hundreds of lines, but never thousands.

Stage 1 – Control of Reading and Printing (marks up to 8/20)

The first version of your program should read the entire input dataset into parallel arrays (or, if you are adventurous, an array of `struct`), counting the data rows as they are read. The heading line should be discarded, and is not required for any of the subsequent processing steps. Once the entire dataset has been read, your program should print the first and last of the meals' details, so that you have confidence that you have read the data correctly. The output of this stage for file `meals0.tsv` must be:

```
mac: ./myass1 < meals0.tsv
S1, 5 meals to be delivered
S1, order 10443, rfd 19:01, distance 2.8 km
S1, order 55432, rfd 19:38, distance 3.2 km
```

Note that the input is to be read from `stdin` in the usual manner, via "<" input redirection at the shell level; and that you must *not* make use of the file manipulation functions described in Chapter 11. No prompts are to be written. You may (and should) assume that the data file contains at most 999 meals; distances should be computed as being straight-line Euclidean in the usual way.

To obtain full marks you need to *exactly* reproduce the required output lines. Full examples can be found on the FAQ page linked from the LMS. You can assume that the input provided to your program will always be sensible and correct, and you do not need to perform any data validation.

You may do your programming in your `grok` playpen, in which case you need to create suitable test files, and execute your program via the 'Terminal" button, see the handout "Running Programs in a Shell Within grok", linked from the "Assignment 1" LMS page. Or you may find it more convenient to move to the `jEdit/gcc` environment. Information about this option is also available on the LMS.

Stage 2 – Simple Sequential Processing (marks up to 16/20)

Ok, now suppose that the drones fly at a constant speed of 17.30 kilometers per hour (yes, that statement should trigger another `#define` into your program), and that it is never windy or rainy (and also that the drones are never attacked by crows or magpies looking for a tasty snack). So order `10443`, which entails a distance of 2.823366 kilometers to the customer, will require $2.823366/17.3 \times 60 = 9.792$ minutes of flight time. For simplicity, all flight times are then taken up to the next largest integer number of minutes (use the function `ceil()` in C's `math.h` library) on a per-journey basis. That is, each of the two flights for order `10443` are calculated as requiring $\lceil 2.823366/17.3 \times 60 \rceil = 10$ minutes.

For simplicity, we will assume that loading and unloading of the drone takes no time. But an allowance of five further minutes (another `#define`) is needed following each trip, to cover recovery of the drone, any cleaning that may be required, and checking/changing of battery packs. Once that is done, the drone is returned to the pool and is available for further service.

Taking all of these components into account, if order `10443` is despatched using "drone A" at its RFD time of 19:01, then the delivery will be made at 19:11. Drone "A" will then arrive back to the kitchen at 19:21, and then return to service (RTS) at 19:26, at which time it is available to commence another delivery. So, if meals are to be despatched as soon as they are ready for delivery, "drone A" can be used again to deliver order `22098`, but not for either of orders `28132` or `34332`. The timings of those two orders require that "drone B" be deployed, and then shortly thereafter, that "drone C" fly into action.

Using the data that your program read during Stage 1, count the peak number of drones that must be in operation to handle the kitchen's deliveries. Label the drones starting with "`A`", then "`B`", and so on. In cases where more than one drone is available you should always allocate the one with the lowest alphabetic label. A new drone (the next available letter) should be deployed only if none of the previously used drones is available at the given RFD time.

According to these rules, for the test file `meals0.tsv` your program should generate the output:

```
S2, order 10443, rfd 19:01, drone A, dep 19:01, del 19:11, rts 19:26
S2, order 28132, rfd 19:07, drone B, dep 19:07, del 19:20, rts 19:38
```

```
S2, order 34332, rfd 19:21, drone C, dep 19:21, del 19:31, rts 19:46
S2, order 22098, rfd 19:26, drone A, dep 19:26, del 19:32, rts 19:43
S2, order 55432, rfd 19:38, drone B, dep 19:38, del 19:50, rts 20:07
S2, a total of 3 drones are required
```

where "dep" indicates the drone departure time from the kitchen; "del" is the delivery time to the customer after the first flight takes place; and "rts" is the "return to service" time after that delivery.

You may assume that at most 26 drones will be required at any given time, labeled "`A`" to "`Z`". You may also assume that the orders that you read in Stage 1 are already sorted in order of RFD time-stamp. Orders are to be processed in the order in which you read them.

Strong hint: Think about using "minutes from midnight" (as total minutes, "mmm") to record all of the timings within your program. Convert "hh:mm" times to that "mmm" format upon input, and then back to "hh:mm" format times upon output. Doing this will make your program quite a bit simpler.

Stage 3 – Better Sequential Processing (marks up to 20/20)

As the kitchen grows, they recruit more staff – kitchen hands, drone wranglers, and so on. It seems that business is booming because of your drone scheduling! Eventually a marketing manager is appointed. They develop an advertising campaign based around the slogan "*from our oven to your door in thirty minutes or less*", promising that every delivery will be completed within thirty minutes of being ready.

After thinking a bit, you realize that this actually gives you more flexibility with the drones, and may even save money. In Stage 2 the scheduling assumption was that every delivery had to be despatched the moment it came out of the kitchen. That meant that order `34332` needed drone "`C`" to be deployed, even though "`A`" was expected back in service in just five more minutes. But under the "within thirty minutes" delivery requirement, order `34332` can be held for five minutes waiting for drone "`A`" to become available. That would mean despatch at 19:26, and delivery at 19:36, still within the thirty minute guarantee "from oven to door".

Add further processing logic to your program to compute the delivery schedule under this new more flexible arrangement (as well as retaining the previous Stage 1 and Stage 2 output). The first drone to return should always be the one pre-allocated to a delayed delivery, with ties (if there are two drones with the same RTS time) again broken by selecting the lowest-labeled drone. If none of the current drones will reach RTS status within a time period that would allow "thirty minutes or less" delivery, a new drone should be deployed and the delivery commenced immediately. Finally, if the actual flight time is thirty minutes or more, the delivery must depart immediately, the same arrangement as in Stage 2. The output from this final stage for the same test file should be:

```
S3, order 10443, rfd 19:01, drone A, dep 19:01, del 19:11, rts 19:26
S3, order 28132, rfd 19:07, drone B, dep 19:07, del 19:20, rts 19:38
S3, order 34332, rfd 19:21, drone A, dep 19:26, del 19:36, rts 19:51
S3, order 22098, rfd 19:26, drone B, dep 19:38, del 19:44, rts 19:55
S3, order 55432, rfd 19:38, drone A, dep 19:51, del 20:03, rts 20:20
S3, a total of 2 drones are required
```

Complete output examples (don't overlook the final line!) are linked from the "Assignment 1" LMS page.

Modifications to the Specification

There are bound to be areas where this specification needs clarification or correction, and you should refer to the "Assignment 1" LMS page regularly and check for possible updates to these instructions. There is already a range of information provided there that you need to be aware of, with more likely to be added.

The Boring Stuff...

This project is worth **20% of your final mark**, and is due at **6:00pm on Friday 30 April**.

Submissions that are made after that deadline will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to take a Health Professional Report (HPR) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops in to something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it with any non-Special Consideration assignment extension requests.

Submission: Programs (your `.c` file) must be submitted **via the LMS**. You may make early "for insurance" submissions if you wish to; only your last submission will be marked. *Don't forget to include and to "sign" and date the Authorship Declaration that is required at the top of your program.*

Testing: You can also carry out pre-submission testing via a system known as `submit`, available at `http://dimefox.eng.unimelb.edu.au` (important: you *must be running one of the University's VPN services in order to access this URL*). You may use `submit` **as often as you want** while developing your program, to get an output comparison; see the separate instructions linked from the assignment LMS page. The `submit` system uses the same computer/compiler that will be employed for post-submission testing (a Unix computer called `dimefox` which is very unforgiving of uninitialized variables and out-of-bounds array accesses). Note that `submit` will likely become congested and non-responsive in the final day or hours before the deadline. arrives. So: *don't leave this assignment to the last minute!*

Marking Rubric: A rubric explaining the marking expectations is linked from the assignment's LMS, and you should study that rubric very closely. Feedback, marks, and a sample solution will be made available approximately two weeks after submissions close.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else, and not developed jointly with anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to "tutoring" sites or online forums, whether or not there is payment involved, and whether or or not you actually employ any solutions that may result, is also serious misconduct. In the past students have had their enrolment terminated for such behavior.

The LMS page contains a link to a program skeleton that includes an Authorship Declaration that you must "sign" and date and include at the top of your submitted program. Marks will be deducted (see the rubric linked from the LMS page) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action, without further warning.

And remember, programming is fun!