SCC.111: HO HO HO - IT'S PANTO SEASON IN SCC

- ASSESSED EXERCISE

THE GOAL

It's week 9 and the first term is nearly behind you. In this week's lab you will begin working on the assessed coursework:

- **You will need to finish this code before the lab next week to enable you to answer the questions in the quiz.**
- **As this is an assessed piece of coursework you will need to make sure this is all your own work.**
- **You will be required to submit your coursework for plagiarism checking.**
- **If you get stuck or need help please ask in your lab session or visit the FAST Hub.**

PROBLEM STATEMENT

It's panto season in SCC (oh no it isn't, oh yes it is) and Santa and the elves have been hard at work – the script is written, the performers selected and the tickets sold. Just one thing remains to be done - the stage lighting needs to be designed. This year Santa has decided the lighting will all be computer controlled.

Your task is to write a program that will control a set of stage lights and bring the SCC panto to life. Santa has designed a new file format (the Lighting Automation File Format (LAFF)). You will need to write a program that is able to parse a file of this format and then execute the instructions it specifies to control the lights.

In stage lighting, individual lights are often referred to as "fixtures". Each fixture will typically have an identifier and a DMX address (an integer between 0 and 255). DMX is the protocol computers use to talk to fixtures. Each fixture will also support a number of channels and the light itself is controlled by setting these channels to a specific a value. For example, for most fixtures seting channel 0 or 1 to the value 255 will set the light to maximum brightness.

A LAFF file consists of four distinct sections that always appear in the following order:

- A set of metatdate about the show (typically its name and date).
- A set of mappings between the fixtures and their DMX addresses.
- A set of cues that describe how the lights should be configured for a specific lighting scene.
- A set of instructions that describe the order in which the cues should be presented and how many times each cue should be repeated.

The elves have made a start on the program to automate the lighting but you need to finish the job for them (they are now busy making toys for Christmas). To do this you need to implement the following four functions:

PART 1: READING IN THE SHOW METADATA

```
int process_show_metadata(FILE *fp, char *showName, char *showDate)
```

The function needs to read the file referenced by *fp* and return the name of the show and the date on which it is to be performed. In LAFF files the show meta data always appears first in the file and is formatted as follows:

```
BEGIN_SHOW_DATA
<the name of the show as a string of less that 80 characters>
<the date of the show as a string of less than 80 characters>
END_SHOW_DATA
```

See the appendix of this coursework for a full example of a LAFF file.

The function should return true (1) if it was able to read the show name and date and false (0) otherwise. You can assume that the person calling the function has allocated sufficient space for `showName` and `showDate` to contain the appropriate information.

PART 2: READING IN THE FIXTURE MAPPINGS

```
int process_fixture_mappings(FILE *fp, int fixtureMappings[])
```

Your function needs to read the file referenced by *fp* and return an array that contains the mapping

between fixture identifiers and their DMX addresses. In LAFF files the fixture mapping always appears second in the file and is formatted as follows:

```
BEGIN_FIXTURE_MAPPING
FIXTURE fixture_id dmx_address
....
....
END_FIXTURE_MAPPING
```

See the appendix of this coursework for a full example of a LAFF file.

The `fixture_id` should be used as an index into the `fixtureMapping` array and the value in the array should be the corresponding fixture's DMX address. For example, if the LAFF file contains:

```
BEGIN_FIXTURE_MAPPING
FIXTURE 3 55
END_FIXTURE_MAPPING
```

Then `fixtureMappings[3] == 55` would be true.

The function should return the number of fixture mappings it was able to read. You can assume that the person calling the function has intitialised the array and that the file may well not contain mappings for every fixture id.

PART 3: READING IN THE SET OF CUES

```
int process_cues(FILE *fp, cueType cues[MAX_CUES][MAX_STEPS])
```

This is the most challenging part of the exercise. Your function needs to read the file referenced by *fp* and initialise a two-dimensional array (`cues`) that contains a set of cues and their associated steps. In LAFF, files cues are contained within a section that begins with BEGIN_CUES and ends with END_CUES. Inside this section one of more cues are specified as follows:

```
BEGIN_CUE <cue id>
FIXTURE <fixture id> <channel_id> <value>
...
...
END_CUE
```

The `cue_id` should be used as an index into the cues array and the steps written into the second dimension of the array for that cue in the order that thet appear. Each element of the array should be of the following type:

```
typedef struct
{
  int id;
  int channel;
  int value;
} cueType;
```

For example, if the LAFF file contains:

```
BEGIN_CUES
BEGIN_CUE 1
FIXTURE 1 1 255
FIXTURE 4 1 255
END_CUE
BEGIN_CUE 2
FIXTURE 1 1 0
END_CUE
END_CUES
```

Then the `cues` array would have the following values:

```
cues[1][0].id == 1
cues[1][0].channel == 1
cues[1][0].value == 255
cues[1][1].id == 4
cues[1][1].channel == 1
cues[1][1].value == 255
cues[2][0].id == 1
cues[2][0].channel == 1
cues[2][0].value == 0
```

The function should *return the number of cues it was able to read*. You can assume that the person calling the function has intitialised the array and that the file may well not contain information for every cue id.

PART 4: EXECUTING THE LIGHTING INSTRUCTIONS

```
int process_steps(FILE *fp, cueType cues[MAX_CUES][MAX_STEPS], int fixtureMappings[])
```

Your function needs to read the file referenced by *fp* and execute the lighting instructions specified. In LAFF files instructions are contained within a section that begins with `BEGIN_PLAY` and ends with `END_PLAY`. Inside this section one of more instructions are specified as follows:

```
BEGIN_PLAY
CUE <cue_id> <in> <out> <repeats>
...
...
END_PLAY
```

The `cue_id` should be used as an index into the cues array. `<in>` and `<out>` are strings but you can ignore them at this stage. The number of repeats specifies how many times the CUE should be exectuted. For each step in the CUE the function should make a call to a function the elves have written called `do_DMX()` :

```
int do_DMX(int address, int channel, int value)
```

The do_DMX() function takes a DMX address, channel and value and passes it onto the relevant fixture.

Once your function has executed all the instructions it should *return the total number of DMX commands* it executed.

Your program should consist of a series of functions as described above. You can use the elves' main program to test your implementation. You will need the functions to answer the quiz questions relating to SCC.111 next week. **You will be required to submit all your functions so that we can run them, and check for plagiarism.**

This is a challenging task so you shouldn't worry if you don't get all the parts finished.

The main program will try and read from a file called `scc111pantolaff.txt` so make sure you have a file of the right type in the same directory as you run your program.

You may find the following references useful.

- A function call `sscanf` that is part of the standard C library (https://man7.org/linux/man-pages/man3/sscanf.3.html)
- You may find you need to remove the `"\n"` character that `fgets()` will read in from the file. You can do this with the following line of code (assuming you have read the line into an array called line) `line[strcspn(line, "\n")] = '\0';`

IMPORTANT NOTES

1. You should have your code complete and tested before next week's lab slot.
2. This is coursework and **thus independent work.** You should not be sharing code or working in teams to solve this problem.
3. If you chose to put your code in a github repo, please make it **private** and *not public*.

APPENDIX A: A SAMPLE LAFF FILE

```
BEGIN_SHOW_DATA
The Great SCC Panto
24.12.2024
END_SHOW_DATA
BEGIN_FIXTURE_MAPPING
FIXTURE 1 27
FIXTURE 2 37
FIXTURE 3 47
FIXTURE 4 57
END_FIXTURE_MAPPING
BEGIN_CUES
BEGIN_CUE 1
FIXTURE 1 1 255
FIXTURE 4 1 255
END_CUE
BEGIN_CUE 2
FIXTURE 1 1 255
FIXTURE 4 1 255
FIXTURE 1 2 255
FIXTURE 4 2 255
FIXTURE 1 2 0
FIXTURE 4 2 0
FIXTURE 1 3 255
FIXTURE 4 3 255
FIXTURE 1 3 0
FIXTURE 4 3 0
END_CUE
END_CUES
BEGIN_PLAY
CUE 1 FADE FADE 1
```

```
CUE 2 CUT CUT 3
END_PLAY
```

SCC.121: STACK

This week you will implement an unbounded stack (using pointers) and apply it to the problem of matching braces. You will already have grasped that in languages, such as C, braces have to match up to delineate scope. So *e.g.*, each of {}, {}{}, {{{}}} is a **matched** string of braces. Each of }{, {}}, and {{} is **ill-matched**. You have to write a program that given a string comprising solely of open and closed braces, determines if it matched or ill-matched.

To enable you need to complete the following task:

- **Task 1** Implement a stack of integers. You are given stack-incomplete.c, which contains driver code to test your stack and some other stuff. You have to implement the push (of which some code is given) and pop functions. Note that *pop* returns data only if stack is not empty; otherwise it returns -1.
- **Task 2** Write pseudocode that illustrates how you will use your stack to solve the matching braces problem.
- **Task 3** Implement the pseudocode in C. You may replace the driver code in main to implement code where you read (using *scanf*) a string from the user and then apply your logic to it.

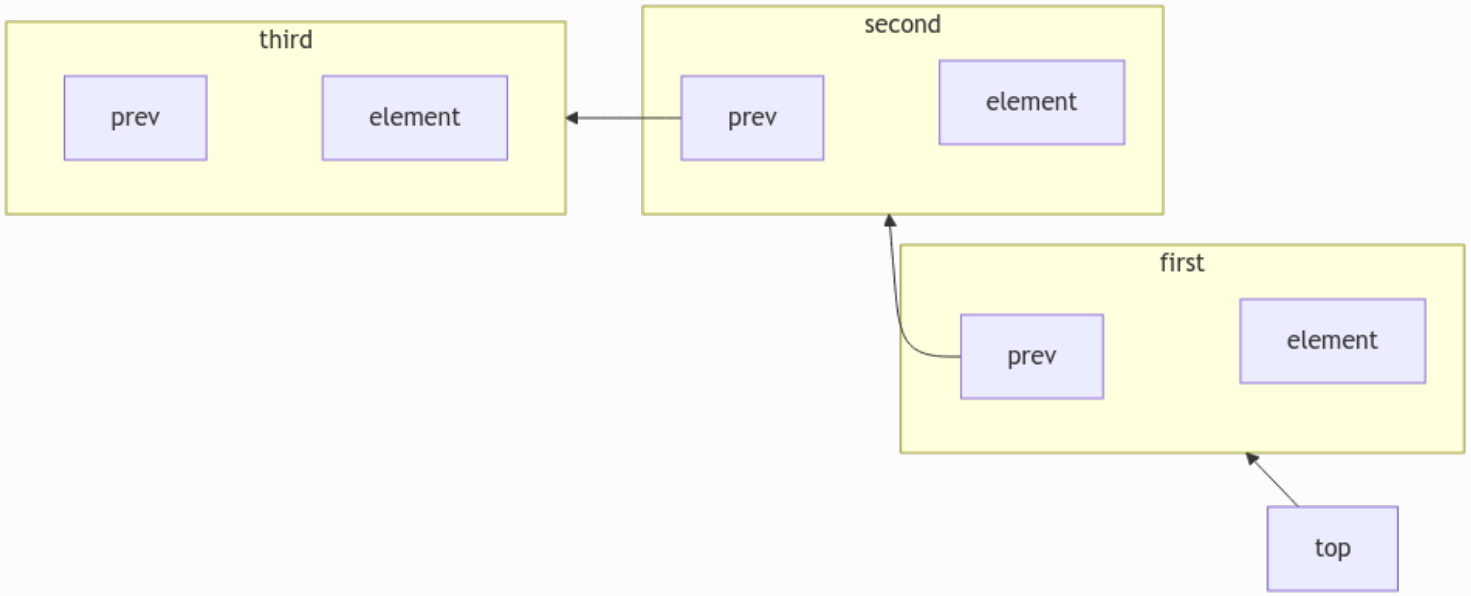

Figure 1: Depiction of a stack with 3 elements using structs and pointer

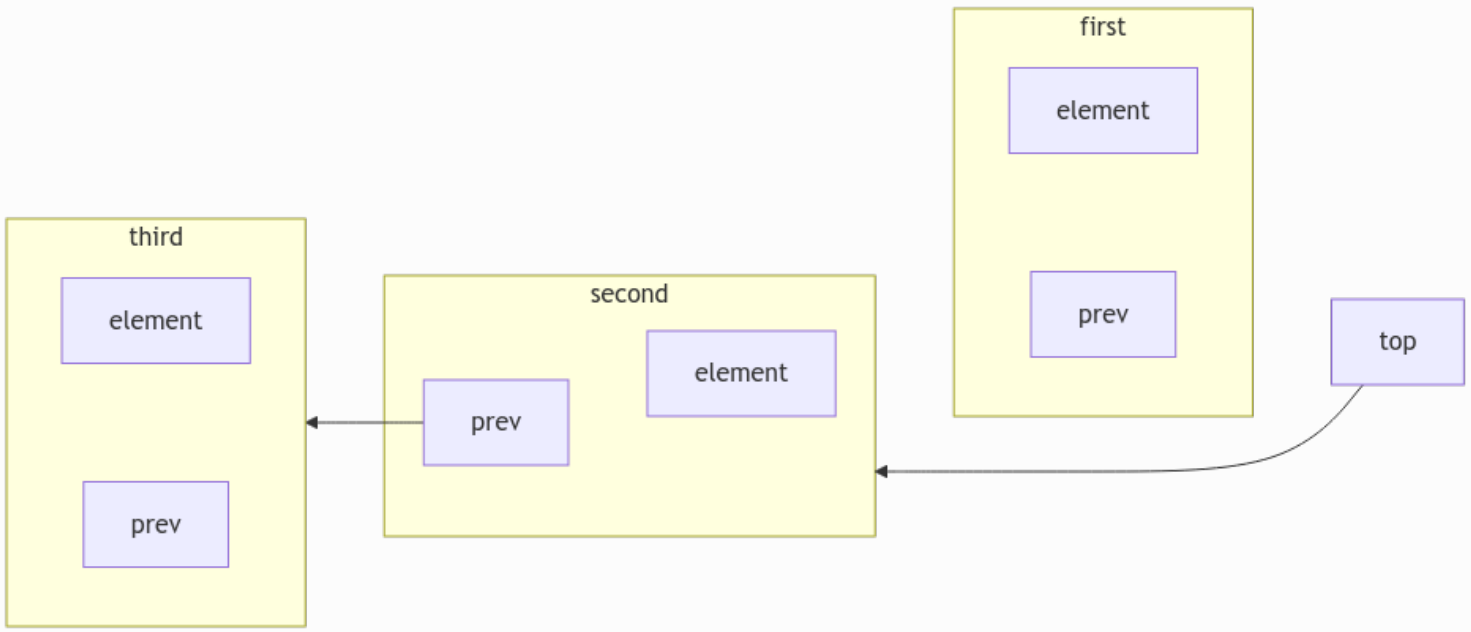

Figure 2: Element pop from stack

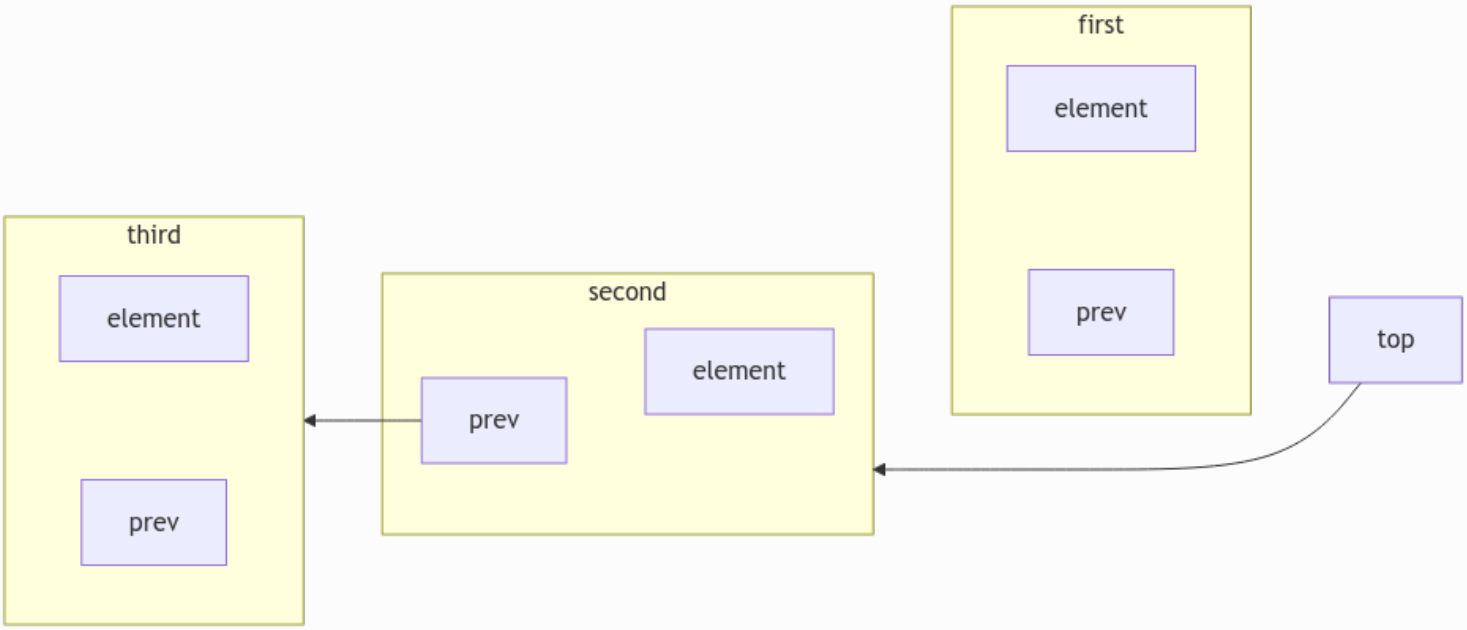

Figure 3: New element push into a stack

SCC.131: CREATING SIMPLE ANIMATIONS AND LOGGING MEASUREMENTS

By the end of this week's SCC.131 lab, you are expected to:

- Develop a basic animation that invokes methods of the `MicroBitDisplay` class and uses composite data types (`struct`).
- Understand and test boundary conditions.
- Revise methods used by the `MicroBitButton` , `MicroBitMessageBus` , `MicroBitThermometer` and `MicroBitLog` classes.
- Combine methods from the aforementioned classes to build an application that logs temperature readings using asynchronous programming (event listeners and event handlers).

TASK 1: TRAPPED SNAKE

Revise the SCC.131 lecture slides of Week 8 (Lecture 1 of 2). Recall that, during this lecture, we developed code for the **animation of a 'dot'**. The code - with comments to help you understand each step - has been included below:

```
#include "MicroBit.h"

MicroBit uBit; // The MicroBit object

int main()
{
    int16_t x = 0; // The x co-ordinate
    int16_t y = 0; // The y co-ordinate

    // Initialise the micro:bit
    uBit.init();
    // Turn on the pixel in location (x, y)
    uBit.display.image.setPixelValue(x, y, 255);
    // Wait for 100 ms
    uBit.sleep(100);

    while (1)
    {
        // Turn off the pixel in location (x, y)
        uBit.display.image.setPixelValue(x, y, 0);

        // Consider the edge conditions, i.e., (4,0), (4,4), (0,4), (0,0)
        // and update the x and y co-ordinates accordingly,
        // so that the moving dot changes direction when it reaches
        // a corner of the 5x5 display.
```

```
        if (x < 4 && y == 0) // The dot moves from the left to the right of the top
        ↪ row
            x++;
        else
        {
            if (x == 4 && y < 4) // The dot moves from the top to the bottom of the
            ↪ right-most column
                y++;
            else
            {
                if (x > 0 && y == 4) // The dot moves from the right to the left of
                ↪ the bottom row
                    x--;
                else
                {
                    if (x == 0 && y > 0) // The dot moves from the bottom to the top
                    ↪ of the left-most column
                        y--;
                }
            }
        }

        // Turn on the pixel in location (x, y)
        uBit.display.image.setPixelValue(x, y, 255);
        // Wait for 100 ms
        uBit.sleep(100);
    }
}
```

Copy and paste the code above onto `main.cpp` in the `\source` folder of the MICROBIT directory (e.g., `microbit-v2-samples`). Your objective is to edit `main.cpp`, such that the display animates a **snake** instead of a dot. Your final code should produce the result shown in this short video when

`MICROBIT.hex` is transferred to micro:bit.

> **To access the video, you may be asked for your university credentials (username and password) to log in.**

For the development of the code, consider the following points:

- Define a **constant** for the length of snake in pixels, e.g., `MAX_LENGTH` . Changing the value of the constant should change the length of the snake in the animation. Note that the length of the snake in the video is 7 pixels.
- Use knowledge acquired from SCC.111 to create a **structure** called `bodypart` with integer members `x` and `y` to store the coordinates of a part of the body of the snake. Essentially, the snake will be an array of type `struct bodypart` .
- Create **functions** that initialize, shift and display the snake on the screen of micro:bit.
- Think of the snake movement in terms of **enqueuing** and **dequeuing** covered in SCC.121 (although you do not need to use pointers). Is the head of the snake the head or the tail of the queue?

TASK 2: TEMPERATURE DATA LOGGER

In the second SCC.131 lecture of Week 8, we discussed about the `MicroBitThermometer` class and the `MicroBitLog` class, which - when combined - can be used to create a temperature data logger. In other words, you can program micro:bit to use its onboard temperature sensor to take temperature readings in regular intervals, and record the readings in a file.

Refer to the SCC.131 lecture slides of Week 8 (Lecture 2 of 2) to recall key functions required for the completion of this task. Open `main.cpp` (found in `/source` of your MICROBIT folder) and develop code that meets the following specifications:

- At startup, micro:bit should be initialised, visibility of the log file should be set to `true` , the logger should be set to record the time of each temperature reading (expressed in seconds), and the sampling period should be set to 2 seconds.
- Listeners should be created for **button A**, the **thermometer** and **button B**.
- When **button A** is clicked, an event handler should:

  - Enable logging if it is currently disabled.
  - Disable logging if it is currently enabled.

    Note: The event handler should just change the value of a variable. The variable should be initialised to 0 (logging is disabled by default) and its value should change every time button A is pressed.

- When the **thermometer** has an update to report (i.e., a new reading every 2 seconds) *and* logging is enabled, an event handler should:

  - Create a new row in the log file and enter the reading in a column labelled `"temperature"`. *There is no need to calibrate the reading.*
  - Change the state of a pixel of the 5x5 display according to the following rules:
    - The very first reading should turn on pixel (0,0).
    - The next pixel in the same row should turn on when a new reading is logged.
    - If all pixels in the same row have been turned on, move to the beginning of the next column, i.e., pixels should turn on from left to right, from the top row to the bottom row. This means that when 25 readings have been logged, all pixels of the display should be on.
    - The 26th reading should turn off pixel (0,0).
    - The same logic as before is followed, but now each reading turns off a pixel. This means that when 50 readings have been logged, all pixels of the display should be off.
    - This process is repeated, i.e., new readings should now turn pixels back on.

- When **button B** is clicked, an event handler should:

  - Clear the display.
  - Clear the contents of the log file.

After you build and transfer `MICROBIT.hex` onto micro:bit, let micro:bit log a few readings and observe the display. Count the number of readings that are being recorded and try to alter the temperature readings, e.g., blow air on the application MCU (be careful not to spit on it!).

When you are satisfied by the number of recorded readings, press button A to stop logging new readings, and safely unmount the micro:bit (i.e., disconnect it from the computer). Wait for a few seconds and plug it back in.

Check the MICROBIT drive and notice that file `MY_DATA.HTM` has appeared. Open it in a browser (double-click on it) to review the readings. Are they as many as you expected? Select **Visual Preview** in the HTML page to view a plot of the temperature as a function of time.

A demo is available for you to watch, but please note that the sampling period has been set to 0.5 seconds (instead of 2 seconds) for the making of the video.

HACKER EDITION

SCC.121: STACKS

Task 1

A mouse is desperately hungry but must traverse a maze to get to a store of cheese. Your job is to write a program that helps the mouse get to the cheese.

The first problem is to represent a maze, which is the input to the problem. We will use a boolean matrix to do that. Suppose our maze is 4X4. Then if cell[i][j] in the maze has a path through it, we will put T (for true) in it; else, it is blocked and we will put F (for false) in it. You may assume that the mouse always enters the maze at cell[0][0] and the cheese is always in cell[3][3]. Different maze instances amount to different combination of values in the cells. **You may hard code the input in your implementation (no need to read if from the user).**

Assume that from a cell, a mouse may move to the cell up, down, left, or right, provided it is not blocked. E.g., given our 4X4 matrix, suppose the mouse is cell[1][1]; then it may move to [0][1] (up), [1][2] (right), [2][1] (down) and [1][0] (left).

> **Beware** – the mouse musn't go outside the maze!

You need a stack because the path the mouse is on may turn out to be deadend and it may have to backtrack to explore another path. The stack is useful for implementing such backtracking.

> **Beware** though not to get stuck in a loop exploring the same path over and over again (don't let our mouse meet the same fate as Mr. Jack Torrance).

Get cracking! Our mouse can wait no more.

Task 2

After our mouse gets to the cheese, it wants to share the path found with other mice. Can you print out the path? If it is too difficult, can you explain the source of the difficulty?

SCC.131: TEMPERATURE DATA LOGGER WITH ADDDITIONAL FEATURES

Think of additional features for the temperature data logger and introduce them in your code.

For instance, another extra feature could be the adjustment of the sampling period. When button A is held down for a long period, the sampling period should drop to 200 ms, i.e., readings should be taken more frequently than before. If button A is held down again, then the sampling period should go back up again to 2 seconds. How would you implement this change? (*Hint*: The listener for button A should be able to detect different events, i.e., a short click and a long click. Consider using a *wildcard*, as explained in the lectures of Week 8).