SCC.111/SCC/121: CATCHING UP

This week there are no new **normal** difficulty tasks for modules SCC.111 and SCC.121. This is an opportunity for students to complete previous weeks' n tasks, especially the implementation of the queue using pointers and to review (Week 7) and catchup with whatever else they have found difficult.

SCC.131: INTRODUCTION TO THE MICRO:BIT RUNTIME ENVIRONMENT

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

- Understand the process of creating a program is C/C++ in the CODAL runtime environment and converting it into a hex file, which can be transferred and executed by micro:bit.
- Use basic methods/functions of the `MicroBit` type and access the display of micro:bit.
- Create images of type `MicroBitImage` and paste them onto the display of the micro:bit at specific coordinates to create eye-catching animations.

HELLO WORLD FOR MICRO:BIT

Component **O**riented **D**evice **A**bstraction **L**ayer (CODAL) is the micro:bit runtime environment for programming the BBC micro:bit in the C/C++ language. CODAL has been developed by Lancaster University. It contains device drivers for all the hardware capabilities of the micro:bit, and also a suite of runtime mechanisms to make programming the micro:bit easier and more flexible.

If you have completed all SCC.131 lab tasks of Week 7, you should have become familiar with the process of flashing code to the micro:bit, i.e., connecting the micro:bit to your computer using the provided USB cable, noticing that micro:bit appears as a USB drive in the file system (called MICROBIT), and copying the generated .hex file over to the MICROBIT drive.

In addition, you should have cloned on your h-drive the repository that contains CODAL (e.g., a folder called **microbit-v2** or **microbit-v2-samples**, depending on how you named it). We shall refer to this folder as the **MICROBIT folder**. Use the **terminal** to go to the **MICROBIT folder**. Type: " `cd source` " to go to the `source` folder and then type " `ls` " to see the contents of that folder. Notice that the folder contains the C++ file " `main.cpp` ". Type " `code main.cpp` " to open and edit the file.

Replace the contents of `main.cpp` with the following lines:

```
#include "MicroBit.h" // library for micro:bit


MicroBit uBit; // create a micro:bit object (similar to a variable)


int main()
{
  uBit.init(); // initialise the micro:bit


  // keep scrolling a message indefinitely
  while(1)
    uBit.display.scroll("HELLO MICRO:WORLD!");
}
```

Notice that standard C libraries, like “ `stdio.h` ”, are not used. Instead, the default “ `MicroBit.h` ” library, which contains methods to access the resources of micro:bit, has been included in the C++ file.

The variable “ `uBit` ” in the code above is an **object** that represents a virtual realisation of your micro:bit. A dot (.) is placed after `uBit` to access a specific resource, e.g. `.display` . A dot is also used when we want to execute a function/method that is associated with micro:bit directly, e.g., `.init` , or with a resource of the micro:bit, e.g., `.dipslay.scroll` .

Save the changes in “ `main.cpp` ”, go back to the terminal and return to the root of the MICROBIT folder, i.e., type “ `cd..` ”. To build the .hex file, type “ `python3 build.py` ” (or “ `python build.py` ” if an older version of Python is installed). Confused? You have written a C++ program but a Python script has been developed to combine your C++ program with the necessary libraries, compile it and build the file `MICROBIT.hex` .

If no errors are raised during the building process, drag and drop MICROBIT.hex to the MICROBIT drive (as explained in the lab sheet of Week 7). Wait until your code is flashed and, if all goes well, the message “HELLO MICRO:WORLD!” should keep scrolling on the display of the micro:bit.

> Congratulations, you have executed your first C++ program on micro:bit!

In summary:

- To progam micro:bit, always edit “ `main.cpp` ” in subfolder “ `\source` ”. When you complete each task in this or future lab sessions, keep operational copies of “ `main.cpp` ” in a personal folder on your h-drive (for safekeeping). Rename the **copies**, e.g., from “ `main.cpp` ” to “ `hellomicroworld.cpp` ”, so that you know what they contain.
- Type “ `python3 build.py` ” in the root of the MICROBIT folder to compile “ `main.cpp` ” and build `MICROBIT.hex` .
- Drag and drop `MICROBIT.hex` to the MICROBIT drive to execute your code (or use WebUSB as an alternative, if you use Google Chrome or Microsoft Edge).

API DOCUMENTATION

Application **P**rogramming **I**nterface (API) is a collection of tools, like variables, functions and methods, which allow programmers to access and control an application. In our case, the API consists of tools included in `MicroBit.h` (and other relevant libraries) that enable us to control and interact with components of micro:bit.

For guidance on how to use the runtime environment in C/C++ and brief descriptions of the semantics and syntax of readily available functions (like `init` and `scroll` , which you have already used) can be found in the online API documentation. Refer to it whenever you want to find out definitions and examples.

LAB TASKS

The key objectives of the SCC.131 lab tasks of Week 8 are to increase your confidence in using the runtime environment and searching the API documentation to find appropriate methods for the needs of your program.

Task 1: Flashing Heart in C/C++

In Week 7, you created a program in MakeCode that shows a flashing heart on the display of micro:bit. Your task today is to create the same program in C/C++.

Open “ `main.cpp` ” in “ `/source` ” and start with the following template:

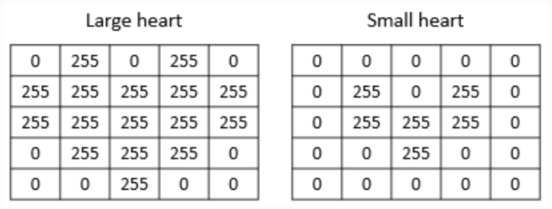
```
#include "MicroBit.h"


MicroBit uBit;


int main()
{
  // Define large heart here
  // Define small heart here


  uBit.init();


  while(1)
  {
    // Develop code for flashing heart here.
  }


}
```

Go to the online API documentation to find out how images can be displayed on the 5x5 screen of the micro:bit. Read the section “Showing Images”.

Notice that `smiley` in section “Showing Inages” of the API documentation is declared as a `MicroBitImage` and is initialised as a **String** using the **String constructor** of the `MicroBitImage`. More specifically, `smiley` consists of five lines (separated by `\n`). Each line consists of five numbers: a “0” implies that the corresponding pixel of the display will not be turned on; a “255” means that the corresponding pixel will be turned on in full brightness.

Based on the example of `smiley`, declare two images of type `MicroBitImage` in the template above. Use the arrays below for guidance on how to initialise the two images:

The next step is to use the `while` -loop in the template to alternate between the two images using the `uBit.display.print()` function followed by `uBit.sleep(500)`. Keep in mind that the input value to `uBit.sleep()` signifies the amount of time in **milliseconds** that micro:bit will pause execution.



Large heart

0	255	0	255	0
255	255	255	255	255
255	255	255	255	255
0	255	255	255	0
0	0	255	0	0

Small heart

0	0	0	0	0
0	255	0	255	0
0	255	255	255	0
0	0	255	0	0
0	0	0	0	0

Figure 1: Hearts presented as arrays

When you finish coding, build MICROBIT.hex and transfer it to the MICROBIT drive (as explained in the section "Hello World for micro:bit" of this lab sheet). The micro:bit should start displaying a flashing heart.

> You created your first images of type `MicroBitImage` and learned how to print them on the display of the micro:bit.

Let us see now how you can move an image on the screen.

Task 2: Moving Smiley

You may have noticed in section "Showing Images" of the online API documentation that a few lines of code have been provided to make a smiley face peep up from the bottom of the screen. The full code is shown below:

```
#include "MicroBit.h"

MicroBit uBit;

int main()
{
  MicroBitImage smiley("0,255,0,255,
↪  0\n0,255,0,255,0\n0,0,0,0,0\n255,0,0,0,255\n0,255,255,255,0\n");

  uBit.init();

  // make your smiley peep up from the bottom of the screen...
  for (int y = 4; y >= 0; y--)
  {
    uBit.display.image.paste(smiley,0,y);
    uBit.sleep(500);
  }
}
```

Copy and paste the code above onto " `main.cpp` ", build MICROBIT.hex and transfer it to the MICROBIT drive. Observe the display of the micro:bit when the transfer is completed.

In general, the function `uBit.display.image.paste(smiley,x,y)` can be decomposed as follows:

- `uBit.display` : Access the display of the micro:bit.
- `image.paste(smiley,x,y)` : Create and show a 5x5 image obtained by pasting `smiley` onto it. The **top-left** corner of `smiley` will be placed at pixel `(x,y)` , i.e., the pixel in **column** `x` and **row** `y` . In the code above, the value of `x` is set to 0. Remember that (0,0) and (4,4) are the coordinates of the top-left and bottom-right corners of the display, respectively.

Edit the code above, so that:

1. `smiley` moves from the **top** of the display to the **bottom**. (*Hint*: The x and y coordinates can take negative values too.)

2. `smiley` moves diagonally from the **top-left** of the display to the **bottom-right** of the display.
3. `smiley` moves diagonally from the **top-left** of the display and **disappears** to the **bottom-right**. Then, re-appears at the **top-right**, moves diagonally and disappears to the **left-bottom** of the display. This process is repeated indefinitely. (*Hint*: you will notice that when you keep pasting `smiley` on the displayed image, a trail will be created, as the image is not cleared (why was a trail not created in the previous two cases?). Use `uBit.display.image.clear()` in the correct lines of your code to clear the image, before `smiley` is pasted onto it.)

> You have mastered basic functions for using the display. You are ready to proceed to the task of the Hacker Edition.

HACKER EDITION

SCC.111/SCC/121: QUEUES

You are familiar with bounded queues, which are extremely useful in implementing synchronization between multiple communication endpoints (threads, actors, parties, whatever). Network routers, the devices that make Internet communication possible use bounded queues for processing and forwarding incoming traffic. When a router receives a network packet, the packet will be stored in a queu, if there is space for it. Packets for the queue are processed from a single very fast processing pipeline, an optimal output port is selected, and the packet is dequeued.

You are given code (in the main() fuction) that simulates a router's behavior. The code asks you to enter a message (assume it is an integer for simplicity). The provided number simulates an incoming message, which is stored in the queue, if there is space. The main function, furthermore, randomly dequeues either 0 or 1 messages. Thus, there is a chance that at some point, the queue may become full, in which case when you try to enqueue a message, it will fail.

Task

For this week's hacker task, you must implement the bounder queue's interface: the methods full(), empty(), enqueue(), dequeue(), and print() (the contents of the queue).

You are already familiar with an array-based bounded queue implementation where everytime you dequeued an item, the rest of the queue is shifted ahead to the beginning of the array. Such shifting is inefficient. A way to avoid this inefficiency is to maintain a circular queue. This is how a circular queue works.

Suppose the queue on which our array is based is of size 4. That means at most four elements can be held in enqueue. Suppose initially the queue is empty, represented here pictorially by [-,-,-,-]

Then enqueueing items (let's assume our items are integers) 20, 40, 60, and 80 in this order, the array is [20, 40, 60, 80]: it is now full.

Dequeuing results in [-, 40, 60, 80]. The head is now pointing to 40, which is not the first element of the array.

Enqueuing 100 results in [100, 40, 60, 80]. Notice the wraparound, which makes the tail of the queue point to 100, which is the first element of the array. So tail is before head in array terms!

Dequeing results in [100,-,60,80]. Dequeing two more times results in [100,-,-,-]

Enqueing 120, 140, 160 in this order results in [100, 120, 140, 160].

Dequeing results in [-,120,150,160] and so on.

You can see how this avoids the inefficiency stemming from shifting; however, the code for the queue is not as straightforward.

Question What happens if you randomly dequeue up to size of queue number of messages? Would that reflect more or less congestion?

In the given code, maximum queue size is 2. You can change it if you like and see what happens.

(The program runs in an infinite loop, as real routers do. Hit Ctrl-C to terminate program.)

HACKER EDITION (SCC.131): REPEATING PATTERN

Your task is to write code that will generate the following repeating pattern on the display of the micro:bit: Watch video. Use your university credentials to log in, if you are prompted to provide a username and a password.

To reverse-engineer the pattern, use these hints and tips:

- An image having dimensions larger than 5x5 is shifted along the x-axis and y-axis of the display. Try to determine the shape that is being shifted and define it is as a `MicroBitImage` using the String constructor.
- You will need to invoke only three functions of `MicroBit.h`. If `uBit` is the declared micro:bit object, the three functions are:
  - `uBit.init()` to initialise micro:bit.
  - `uBit.display.image.paste()` to paste the `MicroBitImage` onto the 5x5 display at the specified coordinates. Check the syntax of this function in Task 2 and the API documentation.
  - `uBit.sleep()` to pause execution for the specified input time in milliseconds, e.g., `uBit.sleep(1000)` will pause execution for 1 sec.
- The use of **five** `for` -loops are required, **four** of which will be within the `while` -loop that repeats the pattern indefinitely.