SCC.111: DICE GAME

OBJECTIVE:

Create a C++ program that models a simple dice game. The game should have a `Dice` object with attributes representing the number of sides on the dice and a method to simulate a roll.

Additionally, create a `Player` class that has a method to guess the outcome of the dice roll. The program should then simulate a round of the game and determine if the player's guess is correct.

PS: Make sure to save your code as a `.cpp` file instead of `.c` .

REQUIREMENTS:

1. Create a Dice class with the following attributes and methods:

   - Attributes:
     - numSides (number of sides on the dice)
   - Methods:
     - roll(): Simulates rolling the dice and returns a random number between 1 and the number of sides.

     For generating a random number you can use the `rand()` function from `stdlib.h` as follows.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

    // Generate a random number between 5 and 25
    int randomNumber = rand() % 21 + 5;

    printf("the random number is: %d\n", randomNumber);
```

```
    return 0;
}
```

To ensure a different random sequence each time, use `srand(time(NULL))` to link the randomness seed to the current timestamp.

```
#include <stdio.h>
#include <stdlib.h> // for rand() and srand()
#include <time.h> // for time()
int main() {

    // Seed the random number generator
    srand(time(NULL));

    int randomNumber = rand() % 21 + 5;
    printf("%d", randomNumber);
    return 0;
}
```

2. Create a Player class with the following attributes and methods:

   - Attributes:
     - name (player's name)
   - Methods:
     - makeGuess(): Allows the player to make a guess for the dice roll outcome.

3. In the main program:

   - Create an instance of the `Dice` class with, for example, 6 sides.
   - Create an instance of the `Player` class.
   - Ask the player to make a guess for the dice roll.
   - Simulate a dice roll using the `roll()` method of the `Dice` class.

- Check if the player's guess is correct and display the result.

SCC121: HASHING/SEARCHING

We talked in class about how complex arithmetic operations such as multiplication and exponentiation may be avoided by using bitwise operations such as shift and or.

1. Assume keys are alphabetical. Let L1, L2, and L3 be alphabetical position of the first three letters in the key, respectively (you may assume keys are at least three letters long). For example, a has position 1 and z has position 26. The arithmetic representation of the hash computation is given by:
   $L1 + L2 * 32 + L3 * (32^2)$
   Write a C function hash_mult that returns the hash given a key and test it. Answer the following questions.

   - What are lowest and highest possible values for hash?
   - The hash represents an index into an array that is the storage for a hash table. Can you spot any glaring problem with computing the hash as above?

2. For the same assumptions as Task 1, implement a C function hash_bitwise that is equivalent to the computation above (refer to the slides). Verify that both hash_mult and hash_bitwise are equivalent.

   - Let bitwise-OR be represented by |. What is $011 + 011$ (these are binary representations)? What is 011|011? Do they produce the same result? Why then is hash_bitwise, which uses | instead of +, equivalent to hash_mult?

3. Implement a C function hash_normalized that uses hash_bitwise and given a key and an array size returns values to locations in the array. We say that hash_normalized is parameterized with respect to the size of the array.

   - Given what we have discussed in class about hash tables, what isthe value of parameterization?

SCC.131: GUESS THE PIXEL!

This is a two-week mini-project that you are expected to complete by the end of Week 12, i.e., you will not be given another SCC.131 lab task in Week 12. Your objective is to develop a two-player game. The first player will be prompted to select one of the 25 pixels of the 5x5 display of the micro:bit. The other player will then have a limited number of attempts to guess the selected pixel. **Although you will develop code for each player, you will have to find a partner to test your code (i.e., play the game), as two micro:bit devices are required.**

STEP 1: PROJECT DESCRIPTION

To get an idea of how the game can be played, a demo has been recorded. During the demo, two micro:bit devices have been placed side by side. The micro:bit on the left belongs to the user who selects a pixel. The micro:bit on the right belongs to the user who attempts to guess the selected pixel. The second user is initially waiting for the first user to select a pixel. This is why letter `W` appears on the display of the second user.

> Watch the demo, and then continue reading the description below. You may be asked to enter your credentials (username and password) to watch the video.

Observe in the video that a cursor appears at the top-left corner of the micro:bit on the left. The user presses button B to move the cursor to the right, from the top row to the bottom row. When button A is pressed, the cursor moves in the opposite direction. If both buttons A and B are pressed together (referred to as button AB thereafter), the pixel that the cursor is on is selected. However, the selection is not final. The user can choose to move the cursor to a different location and press button AB again to select the pixel in that location. To finalise the selection, button AB has to be pressed again while the cursor is on the selected pixel. The location of the selected pixel is transmitted and control is given to the user on the right. Letter `T` appears on the display when transmission has been completed, a sound is played and the selected pixel is shown again on the display of the micro:bit on the left.

The user on the right presses button A or B to move the cursor and button AB to make a guess on the selected pixel (in practice, the two micro:bit devices will not be side by side and the second user will not

know what the first user has selected). Notice that every time the user on the right makes a guess, the respective pixel is lit on the display of **both** micro:bit devices. Therefore, the user on the left knows how close a guess has been, while the user on the right keeps a record of previous guesses. The brightness of the guessed pixels is set to a low level. When the cursor moves over a previously guessed pixel, the guessed pixel should remain lit. If the user has not guessed the pixel after **eight attempts**, a sad face appears (and a sad sound is played) and the game restarts. If the user determines the pixel in fewer than eight guesses, a happy face appears (and a happy sound is played) and the game restarts. Notice that when the user on the right correctly guesses a pixel, the selected pixel on the display of the micro:bit on the left blinks twice.

A key assumption of the project is that wireless communication is **perfect**, that is, information transmitted by a device will always be received by the other device (although this might not be the case in practice).

STEP 2: REQUIREMENTS

Your first task is to create a list of detailed requirements. Given that each of the two micro:bit devices runs a different program, think of key components (variables, functions, features) that are **specific** to each program or are **common** in both programs.

For instance, the two devices cannot be both active or both passive. One device is initially active (for the selection of a pixel), while the other device is passive. When a pixel is selected and its location is transmitted, the corresponding device becomes passive (i.e., pressing the buttons of this device should not have an effect). The other device switches from the passive state to the active state to allow the second user to move the cursor and guess the selected pixel. Therefore, **both programs require a variable to store their current state, i.e., either passive (value 0) or active (value 1)**. Notice that a micro:bit in the passive state is essentially in reception mode, while a micro:bit in the active state is in transmission mode. **Each program also requires a communication mechanism to inform the other program of a state change, so that the other program can move to the opposite state**.

Another common feature that the two programs have is that they both react to buttons A, B and AB, i.e., **both programs require event handlers that are triggered when these buttons are pressed**. Although the functionality of buttons A and B is similar in both programs, the functionality of button AB is different in the two programs.

For example, when **the user who is responsible for selecting a pixel** presses button AB, the appropriate event handler should:

- Check if the program is in the passive or active state.
- If the program is in the active state, check the location of the cursor.
    - If the location of the cursor matches the location of the already selected pixel, the location of the selected pixel should be transmitted and the program should switch to the passive state.
    - If the location of the cursor does NOT match the location of the already selected pixel, the pixel that the cursor is on should be selected - i.e., the coordinates of the selected pixel should be set equal to the coordinates of the cursor. Therefore, the selected pixel will change.

On the other hand, when **the user who is responsible for guessing the selected pixel** presses button AB, the event handler should:

- Check if the program is in the passive or active state.
- If the program is in the active state, check the location of the cursor.
    - If the location of the cursor matches the location of the selected pixel, display a happy face, play a happy sound, reset the game parameters, restart the game and switch to the passive state.
    - If the location of the cursor does NOT match the location of the selected pixel, add the coordinates of the wrongly guessed pixel to an array (so that this pixel is lit) and increment the number of guesses by 1.
    - If the number of guesses is equal to the maximum number of permitted guesses (set to 8 in the demo), display a sad face, play a sad sound, reset the system parameters, restart the game and switch to the passive state.

> The statements above form part of the requirements of your programme, which need to be translated into code. Use the demo and project description in Step 1 to develop a complete list of detailed statements that specify important requirements for the correct operation of your code.

STEP 3: DESIGN AND IMPLEMENTATION

The next step is to convert the requirements into code. Keep in mind that **you need to build two C files**: a C file for the user who will select the pixel and a C file for the user who will attempt to guess the pixel. In other words:

- Create `main.cpp` for the first user, compile it and transfer the `hex` file to your micro:bit device.
- Create a different `main.cpp` for the second user and compile it. If you want to **test the graphical user interface**, you can transfer the `hex` file to your micro:bit device. If you want to **test/play the game**, you need to transfer the `hex` file to a different micro:bit device.
- Keep copies of both files on your h: drive.

> In **week 11**, revise the lab tasks of weeks 8 and 9, and the slides of the lectures in week 8 (both lecture 1 and 2). This revision will help you develop the graphical user interface (GUI) of the game.

> In **week 12**, add the necessary instructions to enable wireless communication between two micro:bit devices. Key instructions will be presented and explained in the first lecture of week 11.

To facilitate development, a template has been provided below. The template can be used as the basis for **both** C files:

```
#include "MicroBit.h"

MicroBit uBit; // The MicroBit object

// --- DECLARE GLOBAL VARIABLES, STRUCTURE(S) AND ADDITIONAL FUNCTIONS (if needed)
↪  HERE ---

// Event handler for the wireless reception of a datagram
void onData(MicroBitEvent e)
{
    // DEVELOP CODE HERE
}

// Event handler for buttons A and B pressed together
void onButtonAB(MicroBitEvent e)
{
```

```
    // DEVELOP CODE HERE
}

// Event handler for button A
void onButtonA(MicroBitEvent e)
{
    // DEVELOP CODE HERE
}

// Event handler for button B
void onButtonB(MicroBitEvent e)
{
    // DEVELOP CODE HERE
}

int main()
{
    // Initialise the micro:bit
    uBit.init();
   // Ensure that different levels of brightness can be displayed
    uBit.display.setDisplayMode(DISPLAY_MODE_GREYSCALE);

    // Set up a listener for the radio.
    uBit.messageBus.listen(MICROBIT_ID_RADIO, MICROBIT_RADIO_EVT_DATAGRAM, onData);

    // Set up listeners for button A, B and the combination A and B.
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_A, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonA);
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_B, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonB);
```

```
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_AB, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonAB);

    // --- ADD LINES TO main() AS REQUIRED ---

    // Enter the scheduler indefinitely and, if there are no other processes running,
    // let the processor go to a power efficient sleep WITHOUT ceasing execution.
    release_fiber();
}
```

If you wish to include sound in the game, as shown in the demo, consider introducing the following instructions in appropriate lines of your code:

- `uBit.audio.setVolume(200); // The higher the value, the louder the sound`
- `uBit.audio.soundExpressions.play("happy"); // Plays a pre-set happy sound`
- `uBit.audio.soundExpressions.play("sad"); // Plays a pre-set sad sound`

You can now start coding! Feel free to include additional requirements or enhance the game (e.g., you could make the GUI more appealing or make the game more complex).

STEP 4: TESTING

Although you can use your own micro:bit to test the GUI of each of the two C files, you need to partner up with another student to test the gameplay.

> You are advised to agree on a test plan with your partner. A student needs to first borrow the micro:bit device and USB cable from their partner to test the gameplay using both micro:bit devices. Then, the partner should be able to gain access to both micro:bit devices and cables to test their code.

When both members of the group (student 1 and student 2) are satisified with their code, they should play the game while testing the following cases:

1. One micro:bit device runs the code for selecting the pixel that was developed by **student 1**, and the other micro:bit runs the code for guessing the pixel that was also developed by **student 1**.

2. One micro:bit device runs the code for selecting the pixel that was developed by **student 2**, and the other micro:bit runs the code for guessing the pixel that was also developed by **student 2**.

3. One micro:bit device runs the code for selecting the pixel that was developed by **student 1**, but the other micro:bit runs the code for guessing the pixel that was developed by **student 2**.

4. One micro:bit device runs the code for selecting the pixel that was developed by **student 2**, but the other micro:bit runs the code for guessing the pixel that was developed by **student 1**.

Cases 3 and 4 imply that requirements have taken into consideration the possibility of using code implemented by different developers without compatibility issues.

HACKER EDITION

SCC.111: DICE GAME++

Build upon the previous exercise and add more complexity to create a multi-round dice game with a scoring system. The game should involve multiple players, each making guesses in each round, and accumulating scores over several rounds.

Requirements:

1. Modify the Player class:
   - Introduce a score attribute to keep track of the player's total score.
   - Implement a updateScore(int points) method to update the player's score - based on the result of each round.
   - Implement a displayScore() method to display the player's current score.
2. Game Setup:
   - Specify the number of players (numPlayers) and the number of rounds (numRounds) at the beginning of the program.
3. Game Logic:
   - In each round, prompt each player to make a guess for the dice roll.
   - Simulate the dice roll using the Dice class.
   - Check each player's guess against the actual roll and update their scores accordingly.
   - Display the results of each round and the total scores after each round.
4. Final Display:
   - At the end of the specified number of rounds, display the final scores for each player.
5. Bonus:
   - Introduce a feature where the player with the highest total score at the end is declared the winner.
   - Implement error handling for player guesses (e.g., ensure guesses are within the valid range).

Rounds running example:

```
// ... (Dice and Player class definitions)

int main() {
    srand(time(NULL));

    int numPlayers = 3;
    int numRounds = 5;

    // ... (Dice and Player instances)
    // put the players in a list
    for (int round = 1; round <= numRounds; ++round) {
        // ... (Round logic)

        // Display scores after each round

        printf("\n--- Scores after Round %d ---\n", round);
        for (int i = 0; i < numPlayers; ++i) {
            players[i].displayScore();
        }
    }

    // ... (Final scores display)

    return 0;
}
```