
Lab 4 — Utilizing Structures in C++

In the previous labs, you have been programming your robot to operate in the real-world. In labs 4 and 5, you will develop the software for keeping track of its position in a virtual world since there are no sensors keeping track of the rooms the robot encounters. Specifically, you will write functions and methods to teach your robot how to enter and study rooms within a virtual representation of the maze. To accomplish your task, this lab includes a digital description of the maze.

Table of Contents

What is New?	2
Data Types	2
C++ Code	2
Creating A New Sketch In Arduino IDE	3
Road Map – A Fork in the Road	5
Take a Step in the Real and Virtual World	5
Initialization of Constants and Variables	5
Defining the EnterRoom block	7
C++ Data Structures	7
Creating the MyRobot Data Structure	8
turnInMaze Function Definition	10
Creating 2-Dimensional turn_table Array	11
stepInMaze Function Definition	12
roomInMaze Function Definition	13
Verifying the operation of the code	13

What is New?

New terminology and concepts are listed below in black. If you have any questions on this information, refer back to the lecture on [C++ Arrays](#).

Data Types

```
const _type           // Define variable as a constant
static _type         // Define variable that will persist after a function return
uint8_t              // Explicit type of a an unsigned 8-bit integer
PROGMEM              // Variable or value will be saved to flash program memory
type *                // Define a pointer of a specific type
struct name {} // Defines a data structure
```

C++ Code

Declarations / Definitions

```
type arrayName[arraySize];           // Array declaration
type functionName(type inputNames...) {funct def}; // Function declaration & definition
type *pointerName;                   // Pointer declaration

struct StructName_t{                  // Structure Prototype
type var1;
type var2;
type varn;                             // etc
}
typedef struct StructName_t TypeName; // Defines a type form structure
```

Bit / Byte Operations

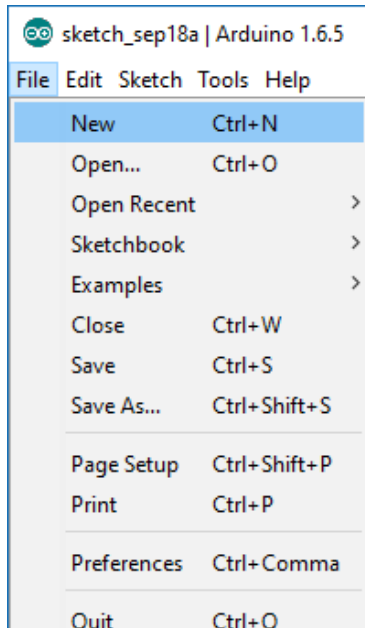
```
variable = byteValue << numberOfShifts; // Left shift byte value by the number defined
variable = _BV (bitNumber);              // Turns a bit number into a byte value
variable |= _BV(bitNumber);              // Set a specific bit in a byte value
variable &= ~_BV(bitNumber);              // Clear a specific bit in a byte value
variable = byteValue | byte Value;       //
```

Displaying Output

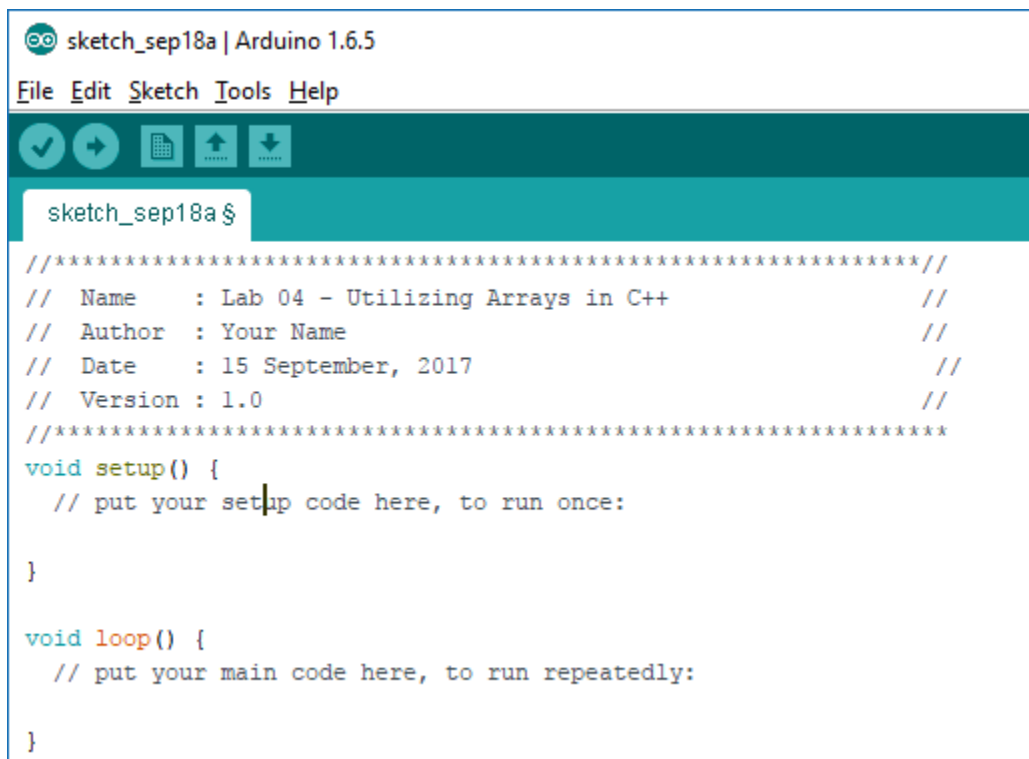
```
Serial.begin(9600);                      //
Serial.print("text or variable");        //
Serial.println("text or variable");       //
```

Creating A New Sketch In Arduino IDE

To start our lab, we will be creating a new sketch in the Arduino IDE. This is done by opening the Arduino IDE and it should generate a blank sketch to start with. The other way is to click File -> New as shown in the figure below (Figure).



The first thing we will be adding into our code is a title block to describe the purpose of the sketch and the author. This will go before the setup function and should look something like this.



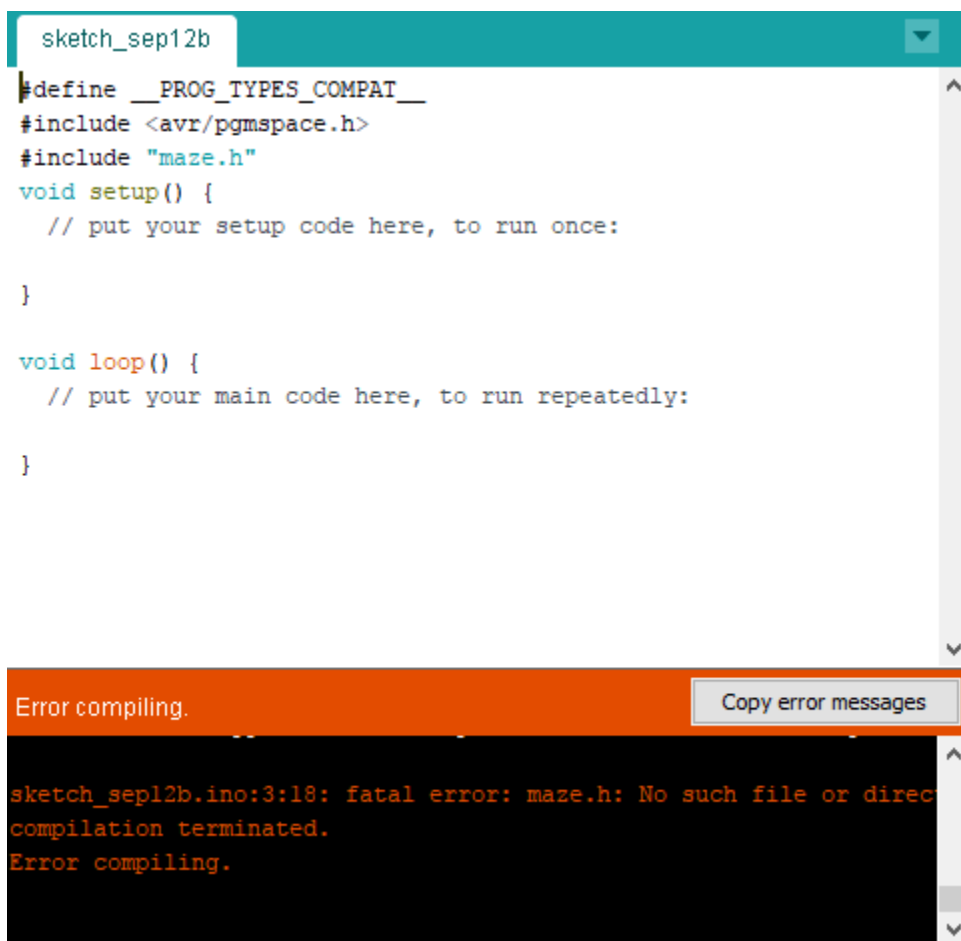
Approved for F'17

Next, you will need to rename the sketch to a more descriptive name. Click on File -> Save and type in the name you want to save it as such as Lab04. It will create a folder for your sketch in the location you have designated for all sketches (Default is in the Arduino folder located in My Documents).

After all of this is done, you will need to include the following lines of code after your title block to incorporate the arrays (look up tables) that you made in Prelab 4.

```
#define __PROG_TYPES_COMPAT__  
#include <avr/pgmspace.h>  
#include "maze.h"
```

Make sure that the maze.h file you created for Prelab 4 is located in the same folder as your Arduino sketch. You may need to reopen the sketch in order for the maze.h file to be shown properly. If you try to compile, it may give you an error as shown below.



```
sketch_sep12b  
#define __PROG_TYPES_COMPAT__  
#include <avr/pgmspace.h>  
#include "maze.h"  
void setup() {  
    // put your setup code here, to run once:  
  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
  
}
```

Error compiling. Copy error messages

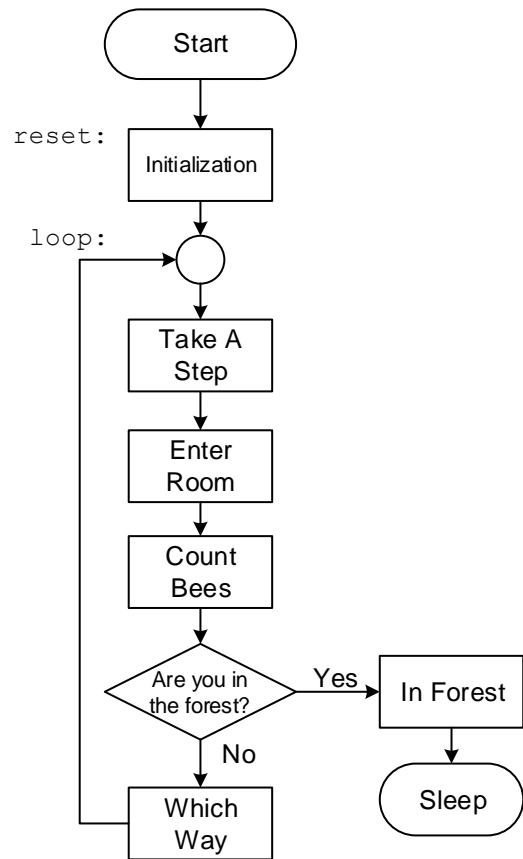
```
sketch_sep12b.ino:3:18: fatal error: maze.h: No such file or directory  
compilation terminated.  
Error compiling.
```

Road Map – A Fork in the Road

In prelab 1, you wrote a program to solve a maze using pseudo instructions: `takeAStep()`, `inForest()`, `enterRoom()`, `seeBees()`, `countBees()`, and `whichWay()`. When you wrote the `whichWay()` routine, you added instructions `turnLeft()`, `turnRight()`, `turnAround()`, plus `hitWall()`, `rightSensor()`, and `leftSensor()`. Now that you have completed Labs 1, 2, and 3, you have taught your robot to follow a line, stop at an intersection, plus turn by calling library subroutines included in “robot3DoT.ino” and your own “pseudo_instr.ino”. In these labs, you have been able to follow your robots progress in the **real world**. In Lab 4 and 5, we will follow our robots progress by creating a **virtual world**. Instructions that help us create and find our way in this virtual world will be located in two new include files. The first is `maze.h` which you can download from the Lab4 folder. This file contains a digital model of the maze. The second is “virtual_instr.ino.”

In this lab, you will be writing the functions needed to implement the psuedo code block:

- `EnterRoom`



Take a Step in the Real and Virtual World

As mentioned in the introduction, your robot currently enters a room within a “real world” maze by taking a step – implemented by the `takeAStep()` function. This lab begins after this first “real world” step by updating our location in our “virtual world” – implemented by three function calls in the `EnterRoom` block. Put another way, until we return execute `EnterRoom`, we do not know where the robot is located within the virtual maze.

Initialization of Constants and Variables

In order to keep track of the robot as it travels in the real world and the virtual world, we will be defining several variables to represent the position, orientation, and action executed. Then, you will be defining the three functions to work with that data. Here is the list of variables and their purposes

<code>dir</code>	Will be one of four values to represent the direction the robot is currently facing
<code>turn</code>	Will be one of four values to represent the action the robot is executing in the real world
<code>row</code>	The current horizontal position within the maze with the top left corner being the origin
<code>col</code>	The current vertical position within the maze with the top left corner being the origin
<code>room</code>	The current room the bear is in based on its position
<code>bees</code>	The number of bees that are at the bear’s current location

Variable **"dir"** is defined by the following truth table.

direction	dir
South	0x00
East	0x01
West	0x02
North	0x03

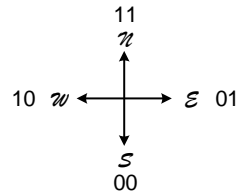


Table 1: Variable "dir" Truth Table

Variable **"turn"** is defined by the following truth table.

action	turn
Do not turn	0x00
Turn Right	0x01
Turn Left	0x02
Turn Around	0x03

Table 2: Variable "turn" Truth Table

Variables **"row"** and **"col"** together locate the robot in the maze. You should have noticed that the row is defined vertically on the far right hand side of the maze.h file and goes from 0x00 (top row) to 0x13 (bottom row). The column (variable col) is defined horizontally along the top and goes from 0x00 (first column) to 0x13 (last column). This can change based on the virtual maze that is defined by the include file.

You will notice that all of these values are in hexadecimal. One reason for this is that it will be familiar for those of you that have taken EE 346. However, it is not required to keep it this way and you can easily use the decimal values instead. For the all of the remaining labs, we will be using the hexadecimal values for all examples and calculations because of the synergy it has with array indexing.

Because there are only 16 possible room combinations for the rooms in the maze and the number of bees will not exceed 15, we can combine them together into the most significant nibble and least significant nibble (four bits) as you did in the prelab. We will be covering how to extract the relevant data from the value stored in the array later on. For now, use the following figure to understand what each room value corresponds to (0000 = 0 = empty room while 1111 = 15 = closed off room).

<input type="checkbox"/> 0000	<input type="checkbox"/> 1000
<input type="checkbox"/> 0001	<input type="checkbox"/> 1001
<input type="checkbox"/> 0010	<input type="checkbox"/> 1010
<input type="checkbox"/> 0011	<input type="checkbox"/> 1011
<input type="checkbox"/> 0100	<input type="checkbox"/> 1100
<input type="checkbox"/> 0101	<input type="checkbox"/> 1101
<input type="checkbox"/> 0110	<input type="checkbox"/> 1110
<input type="checkbox"/> 0111	<input type="checkbox"/> 1111

Figure 2: Wall Definitions

Defining the EnterRoom block

There are three main things that the EnterRoom block needs to do which are updating the direction depending on the action taken, modifying the position based on the direction that the robot moved, and retrieving the new room and bees values from the updated position. Instead of making the enterRoom block into a function that calls three subfunctions, we will be calling those subfunctions from the main loop and handling all of the data there. Before we define these functions, we will discuss C++ Structures and use them as a tool to make the code more readable, organized and intuitive.

C++ Data Structures

Data Structures in C++ provide a greater level of organization for complex systems. Before going into greater details arrays will be reviewed as they give a good insight on how structures work.

for example:

```
type array[size];
```

C++ arrays are a fixed length of a singular data type. This allows us to collect groups of numbers for data for analysis. The issue comes when collections of data are not conducive to arrays or if there is a mixture of data types. Starting from basic C++ foundation, the immediate idea is to make sets of arrays and organize the groups of information by index. That is the core premise of structures! Structures also lays a good foundation as to how Object Oriented Programming (OOP) is organized.

Structure... Structure:

From "What's New" Section:

```
struct StructName_t{                               // Structure Prototype
type  var_1;
type  var_2;
type  var_n;                                     // etc
}
typedef struct StructName_t TypeName;             // Defines a type form structure
```

Structures enable the user to define sets of data and with a variety of data types based on their needs.

For an example:

```
struct Student_t{                                  // Structure Prototype
string  name;                                     // C style string (Char array)
uint8_t age;                                     // integer for age
uint8_t classes[7];                             // array of size 7 for their class list.
float  GPA;                                     // flot with decimals.
}
typedef struct Student_t Student;                 // Defines a type form structure
```

Looking at this example, structures establish a blueprint for how the information you want to store is grouped. The last line with “typedef” sets up for how we can leverage this tool appropriately. Similar to how you can declare more than one instance of any type, we can do the same with this structure now!

For example:

```
int x;
int y;
Student Student1;
Student Student2;
etc.
```

Since we have declared this as a data type we can use this to make multiple instances of this structure, each holding unique information to that struct.

Manipulating the information in the struct requires new notation, the dot (.) operator. This enables us to explicitly change a member within the structure.

It looks like:

```
Student1.age = 21; //or
student1.age = x // assuming x is an uint8_t.
```

Data type rules still hold true. Since age within the structure was defined as an uint8_t, the compiler will check to make sure an uint8_t is placed in that variable.

Finally note how we can choose to say the age of Student2 is different than Student1. This flexibility lays the foundation for how we want to define our Structure in this Lab.

Creating the MyRobot Data Structure

We will now define our struct and assign their values in the program. Our robot has as set of variables that we want to group together to keep track of. From our discussion above we need dir, turn, row, col, room and bees. The generation of this object looks like this:

```
struct MyRobot_t {
    uint8_t dir;
    uint8_t turn;
    uint8_t row;
    uint8_t col;
    uint8_t room;
    uint8_t bees;
}
typedef struct MyRobot_t MyRobot
```

This definition needs to be placed within a header file or else the Arduino IDE will return several errors when we try to use the structure. Place the code above in the maze.h file.

With this structure, we have defined all of our variables as uint8_t but they can be any type. In the main loop, we can create a static structure to hold all of our relevant data.

Wait, we discussed that structs can hold multiple data types but we are holding only one type here. Why not use an array? You can definitely complete this lab with arrays but structs are a better c++ exercise as well as aiding us in making the code more readable. instead of `robot(2) = newTurn;`

The instruction would be `Robot.turn = newTurn;`

Since we need all of our updating functions to have access to the information we will define a static struct inside loop. The struct is now functionally a data type so many of the same rules apply.(const/static/ scope/ etc.) Our Instantiation of the Struct will look like.

```
void loop() {
    static MyRobot Robot{0x03, 0x00, 0x13, 0x00, 0x00, 0x00};
    // more code
}
```

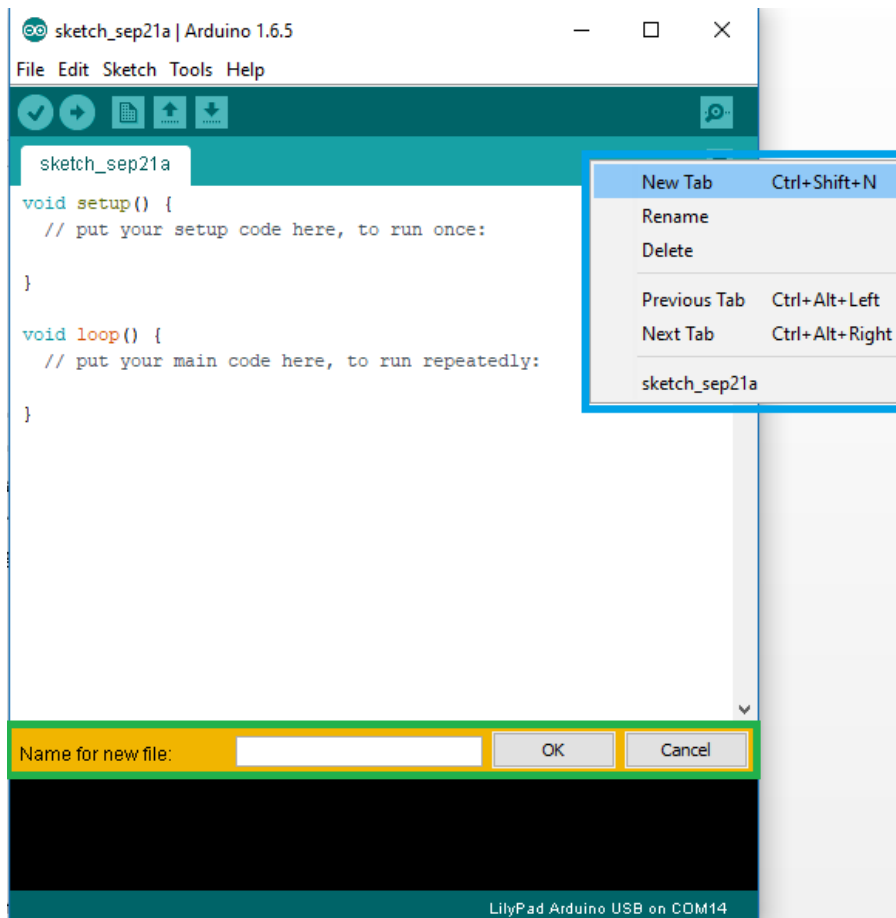
This creates a structure called Robot that initializes the values of each variable to the values listed (similar to an array). The list of default values fills the array in the order it was declared in our header file. So for our case `dir = 0x03`, `turn = 0x00` and etc. If you want to be more rigid in your default setting you may define this in the structure prototype.

In our header...

```
struct MyRobot_t {
    uint8_t dir = 0x03;
    uint8_t turn =0x00;
    .
    .
    .
    etc
```

The final method for doing this is by generating a constructor. This is left to the students to research on what it is doing and how it is implemented. A constructor is a more “true” C++ method of declaring defaults values.

Now that our structure is established we are ready to begin building our functions to change the data inside. We are going to be defining these functions in a new “.ino” file called “virtual_instr.ino”. Click on the drop down arrow in the top right corner and select New Tab as indicated by the blue rectangle in the figure below. Type out the name of the file and click create as shown by the green rectangle.



turnInMaze Function Definition

The objective of the “turn in maze” (`turnInMaze()`) function is to mimic in our virtual world the robot turning in the real world. Our example will be the robot heading south and then turning left. We will be passing the robot structure that contains the current direction and turn values that are used to determine the new direction to face. The function will return an updated structure instead of modifying the `robot.dir` member directly. This is to ensure that the main loop is the only function that has permission to modify the data structure.

Add the following function definition to the “`virtual_instr.ino`” file.

```

uint8_t turnInMaze(uint8_t dir, uint8_t turn){
    // Your code goes here
    return dir
}
  
```

As with most coding problems there are many ways to implement solutions but there is a critical reason for choosing this method. The answer is in the scope and data protection.

By only passing specific portions of our structure we prevent mistakenly changing other values. Using scope ensures that we have full control over what values of our Robot object are changed.

To implement this, our robot object would update direction like this:

```
Robot.dir = turnInMaze(Robot.dir, Robot.turn);
```

Creating 2-Dimensional turn_table Array

The “TurnInMaze” subroutine takes as input the direction the robot **was** facing and any “turn” made. It returns the direction the robot is **now** facing after it took a step in the real world.

The direction the robot is facing is defined in Table 1 and in the figure at right. The turn to be made is defined in Table 2 and duplicated in Figure 4.

00	none
01	right
10	left
11	turn around

Figure 4: Turn

For example, if you are currently facing South (direction = 00₂), and you turn left (turn = 10₂), you will end up looking East (direction = 01₂). Therefore, in the turn left array provided in Table 3, you want the first entry to be East (0b01).

Facing Direction		Direction after turning Left	
	dir.1	dir.0	
	x	y	
South	0	0	East
East	0	1	North
West	1	0	South
North	1	1	West

Table 3: Truth Table for a Left Turn

Now let’s translate what we just learned into the row 10₂ “turn left” in the “turn_table” array found in the “maze.h” file.

As you can see the contents of row 10₂ “turn left” is equal to the last two columns (dir) of Table 3. After filling out the table, you will be able to notice a pattern for how to access the data that needs to be returned. After a little inspection we see that the index address is a function of the turn_val (y-axis) and dir_val (x-axis) as defined by the formula:

```
// Compass    S    E    W    N
// dir        00   01   10   11
const prog_uint8_t turn_table[] PROGMEM =
{ 0b__, 0b__, 0b__, 0b__ // 00 no turn
  0b__, 0b__, 0b__, 0b__ // 01 turn right
  0b01, 0b11, 0b00, 0b10 // 10 turn left
  0b__, 0b__, 0b__, 0b__ // 11 turn around
}
```

Hopefully, you will also remember that to multiply by 2, you only need to shift the contents of a register by 1 place to the left (using the << operator). You can learn more multiplying and dividing by 2 from the C++ Introduction lecture.

In order to access data within the array, you will need to use the following function from the pgmspace library. It has to be done this way because the data is saved in flash program memory by using the PROGMEM variable modifier and we cannot access the array in the normal way (arrayName[index]). With this, we are referring to the starting address of turn_table and modifying it to point to the index value that is desired. If index is equal to 0, it will still get the first element from the array.

```
value = pgm_read_byte_near(turn_table + index);
```

With all of that in mind, add the code below to the turnInMaze() function definition.

```
uint8_t index = (robot.turn << 2) + robot.dir;
uint8_t curr_dir = pgm_read_byte_near(turn_table + index);
return curr_dir;
```

stepInMaze Function Definition

The objective of the “step in maze” (stepInMaze()) function is to mimic in our virtual world the robot taking a step (takeAStep()) in the real world. Our example has the robot heading East in a room at map coordinates 0x13, 0x03 (row, column) and then taking a step. We are passing the robot structure that has the direction and position of the robot *before* it took the step. Using that information, you will be loading data from the map_table[] array that has the values the row and column need to be modified by for the direction the robot was facing. This requires you to add them to the current value of row and col. Because we are trying to limit the access to modifying the values in the robot structure to the main loop, an array is required to pass the new values back to the main loop. Use what you have learned from the Arrays lecture to define the stepInMaze function.

At this point in the lab we will introduce a new structure to help execute this function. It will also introduce another way of utilizing structures.

Define the following struct **above** your Robot_t in your header file

```
struct Coord_t{
    uint8_t row;
    uint8_t col;
};

typedef struct Coord_t Coord;
```

Then put an instance of **Coord** inside the Robot declaration like this:

```
struct My_Robot_t {
    uint8_t dir;
    uint8_t turn;
    uint8_t room;
    uint8_t bees;
    Coord maze;
};
```

Using this new Coord struct we will better group and control our location in the virtual maze.

When making your `stepInMaze()` function we can define the prototype as such:

```
Coord stepInMaze (Coord step){  
    // Update row and col  
    return Coord  
};
```

We will pass the original `Coord` object and then return the new `Coord` object which will update in `Robot`.

The call should look like:

```
Robot.maze = stepInMaze(Robot.maze);
```

This is another great way to use structures (objects) in C++. They give you greater flexibility in sending blocks of related data. Use this method for implementing your `stepInMaze()` function. Remember you can access the information from `step` with `step.row` or `step.col` and perform your indexing from there.

roomInMaze Function Definition

The objective of the “new room in maze” (`roomInMaze()`) function is to provide the new room and bees value based on the robot’s new position for the `whichWay()` function that will determine the next action to take. With the robot structure as an input parameter, the function will be retrieving the room and bees value from the `theMaze[]` array and returning that value. You will need to calculate the index value to access that data correctly. Once it has been extracted, you will need to separate the value to save to the `robot.room` and `robot.bees` members.

Verifying the operation of the code

Now that we have created the structure of the code, we will need to generate some code to test and verify the functionality of our program. Because all of this information is of the “virtual world”, there is no clear way to indicate what the information is. We will be utilizing the serial monitor of the Arduino IDE to display the values of the robot structure after each loop is executed. Within the main loop, you will use the `serial.print()` and `serial.println()` functions to output the values you want to verify. Modify the initial values of the robot structure to test the different inputs that can be sent to the three functions we have defined.