CS203 Lab 5 – Bit-Manipulation

Assigned: Monday September 22nd, 2025
Due: Sunday October 5th, 2025 by 11:59pm.

The Lab Assignment.

You will be creating a text adventure, where the player begins in a starting room and can move from room to room looking for four fish. The names of the fish are: one, two, red, blue. Once you have found the fish you will need to search out the room with the fish pond and release them into the pond.

Throughout this document, you will find links to additional information resources. These links have also been provided on the Moodle page for this lab.

Running the Program.

When the program is run, one of two different options should be provided. The first is the following, which will run the game with a default setting of unsigned integers defined globally in the "`main.c`" source file.

```
./fish_game default
```

The second option will be formulated in the following way, with the second string indicating a file is provided that contains eight rooms and the third string is the name of the input file that will be provided.

```
./fish_game file input_file.txt
```

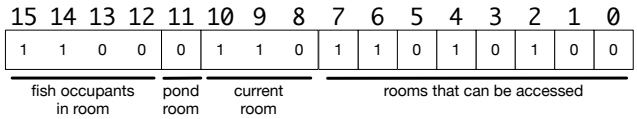
During the initial stages of writing the program, focus on the default option, then once everything is functioning add file-handling for loading the rooms from a file.

Implementation Information.

To make this an interesting project, you will store each room as an "`unsigned short`" value and store all values in an array of eight rooms. To initialize the values, you will use hex values as with the example below. These rooms will be of your own design, but it is also helpful to unit-test your rooms for correctness.

```
unsigned short rooms [8] = { 0x3412, 0x3532, etc... };
```

Each unsigned short value has the following bit-position representation, where each bit-position represents some information.



- Bit-positions 0 through 7 represent the rooms that can be accessed from the current room, with each room indicated by a 1. In the example, the following rooms can be accessed: two, four, six, and seven. These eight bits represent an adjacency list for a given graph node, where the room is also a node. The current room should have its own accessibility bit set to 1.

- Bit positions 8 through 10 give the current room number as a three-bit value. In the example, the presented value is for room six, and accessibility bit six is set to 1.

- Bit position 11 indicates if the current room is the pond room. Only one room can be the pond room, thus only one room will have this bit set to 1.

- Bit-positions 12 through 15 give the fish occupants of that room. There are only four fish in all eight rooms and no single fish can be in more than one room at the same time. Once all four fish have been placed in

the same room where the pond bit is set to 1, then the player has completed the game. No fish should start in the pond room.

- The fish names and bit assignments are:
  - Bit 12 indicates “one fish”
  - Bit 13 indicates “two fish”
  - Bit 14 indicates “red fish”
  - Bit 15 indicates “blue fish”

The goal of this game is to transverse a directed graph using these eight unsigned short values to represent each room. Place fish within this directed graph, and then have the player move the fish around the graph to the pond location. The player will aways start in room 0, and the pond room can be any room that is not room 0.

When a player enters a room, they will be presented with something like the following. This will indicate what options are available to them. The following uses the binary value example above.

```
Room Number: 6

Room Movement Selection:

  0 – move to room 2
  1 – move to room 4
  2 – stay in room 6
  3 – move to room 7

Fish you have : one fish, two fish
Available Fish: red fish, blue fish

Fish Options:

  4 – put down one fish
  5 – put down two fish
  6 – pick up red fish
  7 – pick up blue fish
```

The player can select an action by typing a number and hitting enter. You can find information about basic I/O handling in the book: https://diveintosystems.org/book/C2-C_depth/IO.html#_standard_inputoutput.

If a room movement selection is picked, then the player moves to that room. If the player elects to put-down a fish, then that fish becomes an occupant of that room. If a player elects to pick-up a fish, that fish is no longer an occupant and the player is now carrying it from room to room until they put it down.

Once all fish have been placed in the room containing the pond: then the game should detect this, the player will have finished the game, and the game will exit.

The implementation of this example is tricky, so you might create something simpler. Additionally, you are required to build a hierarchy of functionality across multiple files. Create short functions that can be used by other functions to allow you to leverage the simplicity of the lower level and create higher level simplicity. Each function should do one thing. Thus, it makes sense to create a function to perform a single action, say **`drop_red_fish`**, then use that function in multiple places. If you do this well, our code should be understandable without comments.

It is helpful to create a function that will show you what your rooms look like, this can be used by the unit-testing to display what is going on in your game.

Project Requirements.

For this lab, it is all you. No partial project will be provided, and the requirements are listed here.

File Requirements:

`bit_manip.c bit_manip.h`	These files provide functions for examining information about the rooms and the ability to modify that information. Other than the functions in **`file.c`**, these functions are the only functions that can directly interact with the rooms.
`file.c file.h`	These files provide functionality to load the room data from a file and can directly store the room configurations into the array of unsigned short.
`game.c game.h`	These files provide higher level functionality that implements the game but does not implement the main event loop for playing the game. These functions can directly manipulate the functions in the **`bit_manip`** files.
`main.c`	Main file implements the main event loop driving the game that is implemented in the game files. But this file cannot directly use the functions in the **`bit_manip`** files. This file also uses the file functionality for loading rooms into the game from a provided file. Lastly, this file will provide the default rooms but cannot manipulate this data as described above.

General Requirements:

- The player information should be stored in the **`game.c`** files.
- The room encoding can only be stored in the unsigned short array, generated in the **`main.c`** file.
    - This data cannot be copied out of this array and manipulated separately.
    - The array can only be manipulated directly from the functionality in the **`bit_manip.c`** file.
    - In essence, **`bit_manip.c`** can be seen as a class and has full control of the object data stored within.
- The default unsigned values representing each room will be initially assigned using a hexadecimal value.
- All examination and manipulation of the rooms must be performed using the bitwise operators: and, or, xor, and right/left shifts. (https://diveintosystems.org/book/C4-Binary/bitwise.html)
    - As indicated, all bitwise operations will be implemented in the files "`bit_manip.c`" and "`bit_manip.h`", providing ways of decoding bit values stored in eight "`unsigned short`" values.
- The eight unsigned values will be stored in a global array in the **`main.c`** file, but can be accessed by the functions in **`bit_manip.c`** file using the **`extern`** keyword (described below).
- The **`main.c`** source file should only contain the high-level handling of the main event-loop. There will be a different source file for implementing the game interface called **`game.c`**.
- The **`bit_manip`** and **`game`** functionality needs unit-testing through a main function that is separate from the one provided in the **`main.c`** file.

- The final implementation step should be to provide a mechanism to load room values in from a file as a stored hexadecimal number.
    - A description of file handling is provided by the book in 2.8.3, with details about `fscanf` for input being found here (https://en.cppreference.com/w/cpp/io/c/fscanf).
    - File handling functionality must be implemented source file called **`file_handling.c`** and **`file.h`**

- You cannot use any additional libraries other than Standard I/O as has been done up to this point.

- Remember to unit-test everything that is possible to test. For example, you can unit-test different map configurations if you provide functions for all your actions, instead of fully encoding the game in a big switch statement. Also, having a way to print out the full map is always a handy tool to have for verifying that your map is correct.

- Another helpful tip to remember is to use functions as much as possible to avoid duplicate code.

- Keep things simple and comment when needed.

- You need to provide a **`Makefile`** for compiling your code, as has been see in the previous labs.

Sharing Global Data Between Files.

All c source files, except for the "**main.c**" file, will have a header file as seen in other labs. These files are included in other header files to allow the other associated sources what will be available once the program is fully compiled.

Sharing global data between source files requires you to provide an indication to the compiler that this is happening. For this to work we will use the key word "**extern**", which stands for external, and suggests that a variable is actually defined elsewhere. There will always need to be a single, actual variable definition.

Below is a global variable definition in a "**main.c**" file.

```
#include "bit.h"
unsigned short rooms [3] = { 0x1234, 0x5678, 0x9abc };

int main() {
    test();
}
```

In the above file, there is a header file "**bit.h**" included, that will provide an "**extern**" indicator for a duplicate variable definition "**rooms**". This file is shown below.

```
#ifndef BIT_H
#define BIT_H

#include <stdio.h>
extern unsigned short rooms [3];

void test();
#endif // BIT_H
```

This allows code provided in the "**bit.c**" source file to use the room variable in the "**main.c**" source file, shown below.

```
#include "bit.h"

void test() {
    printf("%x\n", rooms[0]);
    printf("%x\n", rooms[1]);
    printf("%x\n", rooms[2]);
    return;
}
```

So, when the program is run, the following output is produced. Using this technique, global data specified in one place can then be shared everywhere it is redefined using the "extern" key word. Compiling and running the described files should show this result.

```
% gcc -o test main.c bit.c
% ./test
1234
5678
9abc
```