Tutorial – Functions: Zorp part 6

Introduction:

By now your main.cpp file will be starting to look a bit messy. Your file should be around 235 lines of code, and over 200 of those will be the main function alone.

A good rule of thumb to use when writing functions is to never make any function longer than can fit on a screen (perhaps two screens at the most). This is a good way to make sure each function is doing just one thing – which will make your code more modular and reusable.

What we've been doing so far (placing all our code inside one function) is not terribly great practice. It's not your fault, but it's something we need to address. And now that you've formally covered functions in the lecture, now is the perfect time to refactor our code.

In this tutorial we'll break our *main()* function down into several smaller functions.

Requirements:

You will need to have completed the following previous tutorials:

- Zorp Part 1: Variables. Available under the topic *Variables and Constants.*
- Zorp Part 2: Constants. Available under the topic *Variables and Constants.*
- Zorp Part 3: Arrays. Available under the topic *Arrays.*
- Zorp Part 4: Conditional Statements. Available under the topic *Conditional Statements*.
- Zorp Part 5: Loops. Available under the topic *Loops*.

Resources:

There are so many changes we'll need to make in this tutorial that it will be easy to get lost, or perhaps mistakenly place the wrong bit of code somewhere and break the whole game.

While it's good experience for you to attempt to refactor this code on your own, if you do get lost the completed source code from this tutorial will be available on *Canvas* for you to reference.

Global Variables:

As you would be aware, global variables are generally a bad idea… so we're not going to use them.

The one exception to this however is global *constants*.

Previously we had defined a mix of constants as global variables and as local variables inside the *main()* function. To be honest, there was no good reason for separating the declaration of these variables like this, except perhaps for personal preference.

When we had all of our code inside the *main()* function, it didn't matter if we defined the constants locally as all of our code would be able to 'see' those variables anyway. Now that we are breaking our *main()* function down into smaller functions, we're going to need to define these constants globally.

Move all of the constants out of your *main()* function and define them globally as follows:

```
#include <iostream>
#include <windows.h>
#include <random>
#include <time.h>

const char* ESC = "\x1b";
const char* CSI = "\x1b[";

const char* TITLE = "\x1b[5;20H";
const char* INDENT = "\x1b[5C";
const char* YELLOW = "\x1b[93m";
const char* MAGENTA = "\x1b[95m";
const char* RED = "\x1b[91m";
const char* BLUE = "\x1b[94m";
const char* WHITE = "\x1b[97m";
const char* GREEN = "\x1b[92m";
const char* RESET_COLOR = "\x1b[0m";
~~const char* SAVE_CURSOR_POS = "\x1b[s";~~
~~const char* RESTORE_CURSOR_POS = "\x1b[u";~~

const int EMPTY = 0;
const int ENEMY = 1;
const int TREASURE = 2;
const int FOOD = 3;
const int ENTRANCE = 4;
const int EXIT = 5;

const int MAX_RANDOM_TYPE = FOOD + 1;

const int MAZE_WIDTH = 10;
const int MAZE_HEIGHT = 6;

const int INDENT_X = 5;
const int ROOM_DESC_Y = 8;
const int MOVEMENT_DESC_Y = 9;
const int MAP_Y = 13;
const int PLAYER_INPUT_X = 30;
const int PLAYER_INPUT_Y = 11;

const int WEST = 4;
const int EAST = 6;
const int NORTH = 8;
const int SOUTH = 2;
```

Notice we removed the SAVE_ and RESTOR_CURSOR_POS variables (we won't need them anymore), and added a the new MOVEMENT_DESC_Y variable (to specify the line to output the directions the player can move).

The main Function:

I'm going to show you the final *main()* function first.

You probably don't want to delete your current main function just yet – since you'll probably want to copy-and-paste some of the existing code when we write the new functions.

It's probably a bit easier to see all the new functions we'll need to write if we can see (or imagine) what the final *main()* function should look like.

Take a moment to compare this new *main()* function with your current *main()* function. Identify which chunks of code will be moved to each function.

For those that want more of a challenge, try writing each new function yourself before continuing with the rest of the tutorial.

```
void main() {
    // create a 2D array
    int rooms[MAZE_HEIGHT][MAZE_WIDTH];

    bool gameOver = false;
    int playerX = 0;
    int playerY = 0;

    if (enableVirtualTerminal() == false) {
        std::cout << "The virtual terminal processing mode could not
                     be activated." << std::endl;
        std::cout << "Press 'Enter' to exit." << std::endl;
        std::cin.get();
        return;
    }

    initialize(rooms);

    drawWelcomeMessage();

    // output the map
    drawMap(rooms);
```

(continued on next page)

```
    // game loop
    while (!gameOver)
    {
        drawRoomDescription(rooms[playerY][playerX]);

        drawPlayer(playerX, playerY);

        if (rooms[playerY][playerX] == EXIT) {
            gameOver = true;
            continue;
        }
        // list the directions the player can take
        drawValidDirections(playerX, playerY);
        int direction = getMovementDirection();

        // before updating the player position, redraw the old room
              // character over the old position
        drawRoom(rooms, playerX, playerY);

        // update the player's position using the input movement data
        switch (direction) {
        case EAST:
            if (playerX < MAZE_WIDTH - 1)
                playerX++;
            break;
        case WEST:
            if (playerX > 0)
                playerX--;
            break;
        case NORTH:
            if (playerY > 0)
                playerY--;
            break;
        case SOUTH:
            if (playerY < MAZE_HEIGHT - 1)
                playerY++;
        default:
            // the direction was not valid,
            // do nothing, go back to the top of the loop and ask again
            break;
        }
    } // end game loop

       // jump to the correct location
    std::cout << CSI << PLAYER_INPUT_Y << ";" << 0 << "H";
    std::cout << std::endl << INDENT << "Press 'Enter' to exit the program.";
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
}
```

As you read over the new *main()* function, you should be able to see that now it is much easier to read and understand what is happening.

The remainder of this tutorial will discuss each new function in turn.

The enableVirtualTerminal Function:

We've moved the code required to enable the virtual terminal mode into its own function. You'll also note that we've added some error checking so that if enabling this mode fails then the player can be notified.

This is one of the few functions we're adding that will return a value – in this case a *Boolean*.

```
bool enableVirtualTerminal()
{
    // Set output mode to handle virtual terminal sequences
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE)
    {
        return false;
    }

    DWORD dwMode = 0;
    if (!GetConsoleMode(hOut, &dwMode))
    {
        return false;
    }

    dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
    if (!SetConsoleMode(hOut, dwMode))
    {
        return false;
    }
    return true;
}
```

The initialize Function:

The *initialize()* function will contain all the code needed to initialize the game. This will be all the code to populate our map with rooms.

We could have put the virtual terminal initialization code here also, but that would break our 'each function should do only one thing' rule. It would also complicate things a little because if this function ever failed we'd never be sure if it failed because the VT mode couldn't be set, or if there was some other problem with creating our map.

Inside the *initialize()* function we seed the random generator and loop through the rooms array, setting each room to a random type.

```
void initialize(int map[MAZE_HEIGHT][MAZE_WIDTH])
{
    srand(time(nullptr));

    // fill the arrays with random room types
    for (int y = 0; y < MAZE_HEIGHT; y++)
    {
        for (int x = 0; x < MAZE_WIDTH; x++) {
            int type = rand() % (MAX_RANDOM_TYPE * 2);
            if (type < MAX_RANDOM_TYPE)
                map[y][x] = type;
            else
                map[y][x] = EMPTY;
        }
    }

    // set the entrance and exit of the maze
    map[0][0] = ENTRANCE;
    map[MAZE_HEIGHT - 1][MAZE_WIDTH - 1] = EXIT;
}
```

Take special note of how we pass in the 2D array.

Because our array is 2D, we need to hard-code the width and height of the array we pass in. This means we can only ever call this function by passing in an array that is *exactly* these dimensions.

This differs slightly to how we pass in a 1D array by also passing in its length (covered in the lecture).

There is a way to do something similar with a 2D array, so that we could pass in an array of any size along with its width and height, but we won't cover this till a later session of pointers.

Since our array is set to a constant size at compile time, this is a perfectly acceptable way to pass our array to a function.

Note that just like with 1D arrays, anything we do to the values in the 2D array persists when we exit the function. So after we exit the *initialize()* function, all the rooms in the array will hold the values we set them to within the function.

The drawWelcomeMessage Function:

```
void drawWelcomeMessage()
{
    std::cout << TITLE << MAGENTA << "Welcome to ZORP!" << RESET_COLOR
              << std::endl;
    std::cout << INDENT << "ZORP is a game of adventure, danger, and low
              cunning." << std::endl;
    std::cout << INDENT << "It is definitely not related to any other
              text-based adventure game." << std::endl << std::endl;
}
```

There's nothing fancy going on here. We're simply outputting the welcome message to the console.

You may ask 'why would we make a new function for just 3 lines?'.

Doing this means that if we wanted to change the welcome message later, we know exactly where this code lives. We could also make future changes (perhaps change *where* the message is drawn on the console, or add other logic) without affecting the rest of the program.

Adding functions is sometimes a trade-off between future-proofing our program and writing a bit more code. At a minimum, it makes everything neater and more readable.

The drawMap and drawRoom Functions:

In our drawMap function, rather than simply paste our exiting code into a function, we'll break it down even further. Within our drawMap() function, any time we want to draw a room, we'll call the *drawRoo()* function and pass in the type of room we want to draw.

The reason we want to do this is because later in our game we need to redraw the current room (so we can remove the player's old marker). Because we need to call code that will draw a room in two places (when drawing the whole map, and when redrawing an individual room), it makes sense to create a single function we can call in both cases.

To make the *drawRoom* flexible enough to be called in both cases, it will need to accept the *x,y* indices of the room in the array and calculate the correct column and row on the console where the room will be drawn. Below is the code for these two functions:

```
void drawRoom(int map[MAZE_HEIGHT][MAZE_WIDTH], int x, int y) {
    // find the console output position
    int outX = INDENT_X + (6 * x) + 1;
    int outY = MAP_Y + y;

    // jump to the correct location
    std::cout << CSI << outY << ";" << outX << "H";
    // draw the room
    switch (map[y][x]) {
    case EMPTY:
        std::cout << "[ " << GREEN << "\xb0" << RESET_COLOR << " ] ";
        break;
    case ENEMY:
        std::cout << "[ " << RED << "\x94" << RESET_COLOR << " ] ";
        break;
    case TREASURE:
        std::cout << "[ " << YELLOW << "$" << RESET_COLOR << " ] ";
        break;
    case FOOD:
        std::cout << "[ " << WHITE << "\xcf" << RESET_COLOR << " ] ";
        break;
    case ENTRANCE:
        std::cout << "[ " << WHITE << "\x9d" << RESET_COLOR << " ] ";
        break;
    case EXIT:
        std::cout << "[ " << WHITE << "\xFE" << RESET_COLOR << " ] ";
        break;
    }
}
```

```
void drawMap(int map[MAZE_HEIGHT][MAZE_WIDTH])
{
    // reset draw colors
    std::cout << RESET_COLOR;
    for (int y = 0; y < MAZE_HEIGHT; y++)
    {
        std::cout << INDENT;
        for (int x = 0; x < MAZE_WIDTH; x++) {
            drawRoom(map, x, y);
        }
        std::cout << std::endl;
    }
}
```

You will want to make sure you define the *drawRoom()* function *before* the *drawMap()* function, otherwise you'll need to declare the function before the first time it is used. Declaring functions (and the reason we may need to declare functions) was covered in the lecture.

The drawRoomDescription Function:

This function outputs a message depending on the type of room the player is currently in.

The logic is relatively simple. We just need to move the cursor to the correct location on the console, check the room type passed into the function, and output the message for that room type.

```
void drawRoomDescription(int roomType)
{
    // reset draw colors
    std::cout << RESET_COLOR;
    // jump to the correct location
    std::cout << CSI << ROOM_DESC_Y << ";" << 0 << "H";
    // Delete 4 lines and insert 4 empty lines
    std::cout << CSI << "4M" << CSI << "4L" << std::endl;

    // write description of current room
    switch (roomType) {
    case EMPTY:
        std::cout << INDENT << "You are in an empty meadow. There is nothing of
            note here." << std::endl;
        break;
    case ENEMY:
        std::cout << INDENT << RED << "BEWARE." << RESET_COLOR << " An enemy is
            approaching." << std::endl;
        break;
    case TREASURE:
        std::cout << INDENT << "Your journey has been rewarded. You have found some
            treasure." << std::endl;
        break;
    case FOOD:
        std::cout << INDENT << "At last! You collect some food to sustain you on
            your journey." << std::endl;
        break;
```

(continued on next page)

```
    case ENTRANCE:
        std::cout << INDENT << "The entrance you used to enter this maze is
            blocked. There is no going back." << std::endl;
        break;
    case EXIT:
        std::cout << INDENT << "Despite all odds, you made it to the exit.
            Congratulations." << std::endl;
        break;
    }
}
```

The drawPlayer Function:

This is another simple function. Using a bit of simple math, we calculate the correct position on the console to draw the player, and then write the player's marker character to the screen.

```
void drawPlayer(int x, int y)
{
    x = INDENT_X + (6 * x) + 3;
    y = MAP_Y + y;

    // draw the player's position on the map
    // move cursor to map pos and delete character at current position
    std::cout << CSI << y << ";" << x << "H";
    std::cout << MAGENTA << "\x81" << RESET_COLOR;
}
```

The drawValidDirections Function:

Because our map is square and has no walls, we can determine all the valid moves the player can make using only the player's current position and the dimensions of the map.

The map's with and height (MAZE_WIDTH and MAZE_HEIGHT) constants are global now, so we only need to pass the player's x and y position into this function.

The function will move to the correct location on the console, then output the valid movement directions according to where the player is on the map. Note the use of ternary statements.

```
void drawValidDirections(int x, int y) {
    // reset draw colors
    std::cout << RESET_COLOR;
    // jump to the correct location
    std::cout << CSI << MOVEMENT_DESC_Y + 1 << ";" << 0 << "H";
    std::cout << INDENT << "You can see paths leading to the " <<
        ((x > 0) ? "west, " : "") <<
        ((x < MAZE_WIDTH - 1) ? "east, " : "") <<
        ((y > 0) ? "north, " : "") <<
        ((y < MAZE_HEIGHT - 1) ? "south, " : "") << std::endl;
}
```

The getMovementDirection Function:

The final function we need to write is the function that retrieves and validates the player's movement input.

The code is very similar to what we have previously been doing.

First, we jump to the correct position on the console and output the message "where to now?". Then the cursor jumps to the input location and waits for player input. Finally, we validate that an integer was input and return the result.

Notice how we don't validate the input to make sure it's one of the NORTH, SOUTH, EAST, or WEST values. We simply return whatever was entered. If there was an error (for example, the player didn't enter an integer) we simply output 0.

The calling function will decide what to do if an invalid integer was returned by this function.

```
int getMovementDirection()
{
    // jump to the correct location
    std::cout << CSI << PLAYER_INPUT_Y << ";" << 0 << "H";
    std::cout << INDENT << "Where to now?";

    int direction;
    // move cursor to position for player to enter input
    std::cout << CSI << PLAYER_INPUT_Y << ";" << PLAYER_INPUT_X << "H" << YELLOW;

    // clear the input buffer, ready for player input
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());

    std::cin >> direction;
    std::cout << RESET_COLOR;

    if (std::cin.fail())
        return 0;
    return direction;
}
```

That's it! You should now have several new functions in your program along with an updated *main()* function. Hopefully you can see the value in breaking your code up into functions, and taking a bit of time to try to make those functions as reusable as possible.

Don't forget, if you got lost or your program isn't working, you can grab the completed tutorial from the *Resources* section on *Canvas*.