Tutorial – Loops: Zorp part 5

Introduction:

In this tutorial we start getting to the really exciting stuff. We're going to implement a game loop, and have a playable character walk around our game map.

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*.

The Game Loop:

A game loop is exactly what it sounds like. Typically implemented as a *while* loop, it keeps looping and executing our game logic until the *game over* condition is met.

In our text-based game, we'll use it to keep getting directions from the player until they reach the exit room.

Not everything goes inside the game loop. Any initialization or clean-up code should remain outside the loop as it is only executed once. In our case, our welcome message and the 'press enter to exit' message will both be placed outside the loop.

The remaining code will need some slight adjustment before it can be placed inside the body of our game loop.

Mostly, this will simply involve making sure we reset the cursor to the correct position on the console so that we can remove or overwrite any existing text with the new output text.

Also, we'll remove the questions that ask for the player's height and name and replace it with a description for the room followed by a question asking where the player would like to move. The player will then be able to input their desired direction of movement and the game will update accordingly.

Update the *main.cpp* file as follows (unless specified, all other code should remain the same):

```
void main() {
    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 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;

...
```

We'll first define some more constant variables. The first set will be used to position the output on the console:

- INDENT_X: how many spaces to use to indent all text. This can also be used as a tab.
- ROOM_DESC_Y: the line to use for our room descriptions.
  Each type of room will have a description. For example, for the empty room the description will be "You are in an empty meadow. There is nothing of interest here."
- MAP_Y: the first line where the map is drawn.
- PLAYER_INPUT_X: the character column where the player will type their input.
- PLAYER_INPUT_Y: the line where the player will type their input.

The next four constants are values are directions in which the player can move. Rather than have the player type in the direction, we've used the numbers on the keypad. This will make it easier for us to validate the input and process it, but you could just as easily use the 'wsad' keys.

Next, we want to remove the height question and update our map output using the constant position values we defined. Continue updating the next section of code as follows:

```
    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;

       std::cout << INDENT << "First, some questions..." << std::endl;

    // save cursor position
    std::cout << SAVE_CURSOR_POS;

    // output the map
    std::cout << std::endl;
    std::cout << std::endl;
    std::cout << std::endl;
    std::cout << std::endl;

    std::cout << CSI << MAP_Y << ";" << 0 << "H";
    for (int y = 0; y < MAZE_HEIGHT; y++)
    {
       std::cout << INDENT;
       for (int x = 0; x < MAZE_WIDTH; x++) {
        switch (rooms[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;
           }
        }
       std::cout << std::endl;
    }
...
```

That's it for the initialization code. The next section of updates relates to everything inside the game loop.

Before we add the game loop, ***remove*** the following code:

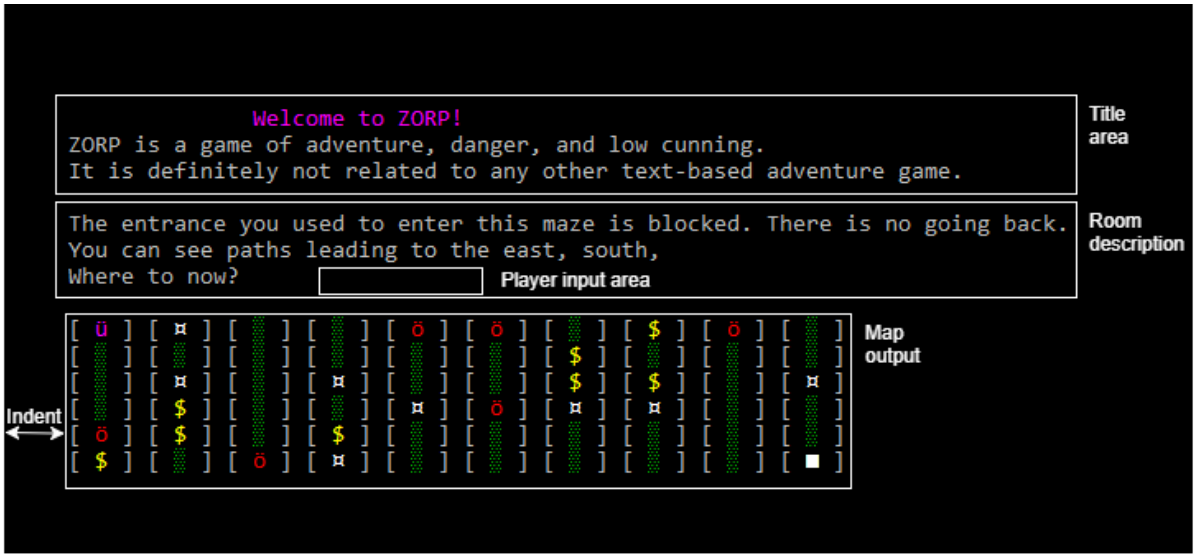
```
    std::cout << RESTORE_CURSOR_POS;  // move the cursor back to the top of the map
    std::cout << INDENT << "How tall are you, in centimeters?"<<INDENT<<YELLOW;
    std::cin >> height;
    std::cout << RESET_COLOR << std::endl;
    if (std::cin.fail()) {
        std::cout << INDENT << "You have failed the first challenge and are eaten
            by a grue.";
    }
    else {
        std::cout << INDENT << "You entered " << height;
    }
    // clear input buffer
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();

    std::cout << RESTORE_CURSOR_POS;  // move cursor to start of the 1st question
    std::cout << CSI << "3M";     // delete the next 3 lines of text
    std::cout << CSI << "3L";     // insert 3 lines (so map stays in same place)

    std::cout<<INDENT<<"What is the first letter of your name? "<<INDENT<<YELLOW;

    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin >> firstLetterOfName;
    std::cout << RESET_COLOR << std::endl;

    if (std::cin.fail() || isalpha(firstLetterOfName) == false) {
        std::cout << INDENT << "You have failed the second challenge and are eaten
            by a grue.";
    }
    else {
        std::cout << INDENT << "You entered " << firstLetterOfName;
    }
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();

    // move cursor to start of the 1st Q, then up 1, then delete and insert 4 lines
    std::cout << RESTORE_CURSOR_POS << CSI << "A" << CSI << "4M" << CSI << "4L";
    if (firstLetterOfName != 0) {
        avatarHP = (float)height / (firstLetterOfName * 0.02f);
    }
    else {
        avatarHP = 0;
    }
    std::cout << INDENT << "Using a complex deterministic algorithm, it has been
        calculated that you have " << avatarHP << " hit point(s)." << std::endl;
```

That's most of our current program removed. We'll be replacing this with something more game-like.

To insert the game loop, we start by defining the *gameOver* variable, then jump straight into the loop itself:

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

    // game loop
    while (!gameOver)
    {
        // prepare screen for output
        // move cursor to start of the 1st Q, then up 1, delete and insert 4 lines
        std::cout << RESTORE_CURSOR_POS << CSI << "A" << CSI << "4M" << CSI << "4L"
            << std::endl;

        // write description of current room
        switch (rooms[playerY][playerX]) {
        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 << "BEWARE. 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;
        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;
            gameOver = true;
            continue;
        }

        // list the directions the player can take
        std::cout << INDENT << "You can see paths leading to the " <<
            ((playerX > 0) ? "west, " : "") <<
            ((playerX < MAZE_WIDTH - 1) ? "east, " : "") <<
            ((playerY > 0) ? "north, " : "") <<
            ((playerY < MAZE_HEIGHT - 1) ? "south, " : "") << std::endl;

        std::cout << INDENT << "Where to now?";

...
```

After starting the game loop, the first thing we do is set our cursor to the position on the console where the room descriptions will be written, then we delete the next four lines. When we delete lines of text, we also need to insert new empty lines, or else the position of our map will change. So we do that too.



This screenshot should give you an indication of how we've broken down the areas on our console.

In the following line of code (extracted from the modifications above), we're passing a few different commands to the console:

```
std::cout<<RESTORE_CURSOR_POS<<CSI<<"A"<<CSI<<"4M"<<CSI<<"4L"<<std::endl;
```

The commands break down like this:

- RESTORE_CURSOR_POS: this is the command "\x1b[u". It restores our previously saved cursor position. We saved the cursor position after outputting the "Welcome to Zorp…" message.
- `CSI << "A"` : this is the command "\x1b[A". It tells the cursor to move one line up.
- `CSI << "4M"` : this is the command "\x1b[4M". It deletes 4 lines of console text, starting from the cursor's current location. We can delete any number of lines of text by specifying a different value.
- `CSI << "4L"` : Similar to the delete command, this command inserts 4 empty lines of text at the cursor's location. This is necessary so the position of the map remains the same.

The last part of code will output the directions the player can move in. If we wanted to be more verbose, we could have used the following code instead:

```
std::cout << INDENT << "You can see paths leading to the ";
if (playerX > 0)
    std::cout << "west, ";
if (playerX < MAZE_WIDTH - 1)
    std::cout << "east, ";
if (playerY > 0)
    std::cout << "north, ";
if (playerY < MAZE_HEIGHT - 1)
    std::cout << "south, ";
std::cout << std::endl;
```

Instead, we've used ternary operators to condense the code a little. Although you are free to use whichever style you prefer, you should be familiar ternary operators.

Before we handle the player's input, we need to update the map using player's current position.

The player character will be represented on the map using the character ü, drawn in magenta. This character was chosen due to its resemblance to a smiling face.

To update the player's position on the map, we simply output this character over the top of the existing room character. This is done by calculating the exact x and y coordinate of the room's output character using the player's current coordinates.

The following code achieves this:

(All of this code is new, and should be inserted after the previous update ending with the message "where to now?").

```
        int x = INDENT_X + (6 * playerX) + 3;
        int y = MAP_Y + playerY;

        // 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";

        // 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());

        int direction = 0;
        std::cin >> direction;
        std::cout << RESET_COLOR;

        if (std::cin.fail())
            continue;          // go back to the top of the game loop and ask again
```

This code calculates where to draw the player's character marker, outputs it to the console (overwriting the room's character at that location), and finally accepts the player's next move (the program has already previously asked the player to enter their next move).

The last chunk of code we need to write handles the processing of the player's input. First, we remove the player's marker character by overwriting it with the current room's character. (We need to do this *before* we update the player's position, since we're not storing the player's last position).

The following update will remove the player's marker. (All of this code is new, and continues from the end of the previous update):

```
        // before updating the player position, redraw the old room character over
        // the old position
        std::cout << CSI << y << ";" << x << "H";
        switch (rooms[playerY][playerX])
        {
        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;
        }
```

Since we've already calculated the player marker's x and y position, we can reuse those values.

Finally, update the player's *x* and *y* position according to the input value. (All of this code is new, and continues from the end of the previous update):

```
        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:
            // do nothing, go back to the top of the loop and ask again
            break;
        }

    } // end of game loop
...
```

That's it! That is all our updates needed to insert a game loop into our game.

If you execute the game now, you should be able to use the number pad to navigate your player around the map.

Once you reach the exit room, your game over condition will be met and the program will end.