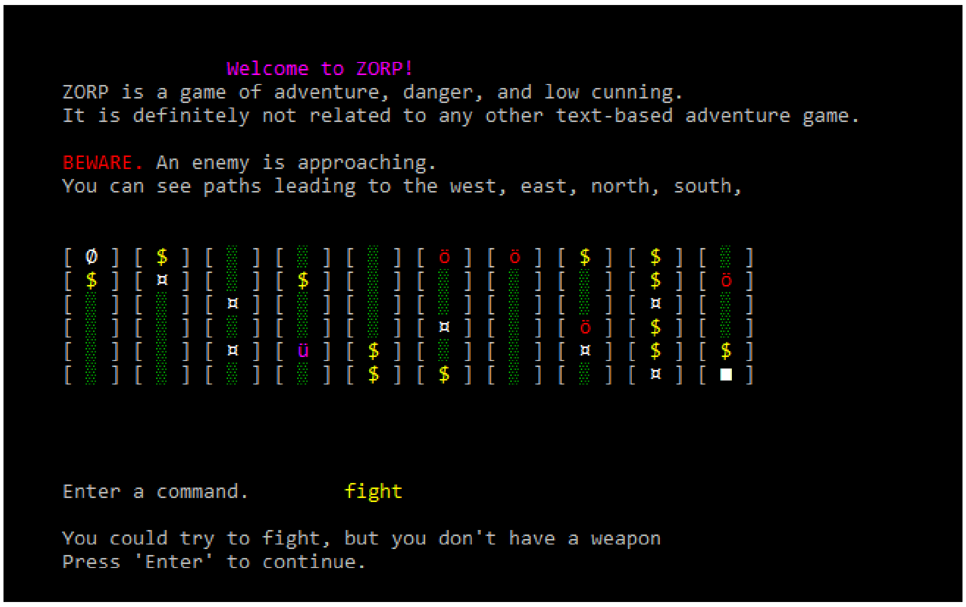
Tutorial – Character Arrays: Zorp part 7



```
              Welcome to ZORP!
ZORP is a game of adventure, danger, and low cunning.
It is definitely not related to any other text-based adventure game.

BEWARE. An enemy is approaching.
You can see paths leading to the west, east, north, south,

[ Ø ] [ $ ] [ ░ ] [ ░ ] [ ░ ] [ ö ] [ ö ] [ $ ] [ $ ] [ ░ ]
[ $ ] [ ¤ ] [ ░ ] [ $ ] [ ░ ] [ ░ ] [ ░ ] [ ░ ] [ $ ] [ ö ]
[ ░ ] [ ░ ] [ ¤ ] [ ░ ] [ ░ ] [ ░ ] [ ░ ] [ ░ ] [ ¤ ] [ ░ ]
[ ░ ] [ ░ ] [ ░ ] [ ░ ] [ ░ ] [ ¤ ] [ ░ ] [ ö ] [ $ ] [ ░ ]
[ ░ ] [ ░ ] [ ¤ ] [ ü ] [ $ ] [ ░ ] [ ░ ] [ ¤ ] [ $ ] [ $ ]
[ ░ ] [ ░ ] [ ░ ] [ ░ ] [ $ ] [ $ ] [ ░ ] [ ░ ] [ ¤ ] [ ■ ]



Enter a command.        fight

You could try to fight, but you don't have a weapon
Press 'Enter' to continue.
```

Introduction:

Currently users can only enter movement commands via the number pad, with each number corresponding to a movement direction.

In this tutorial, we will make changes to our game to allow players to type a sentence command. Our game will then look for keywords within the player's input and determine which command to execute.

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*.
- Zorp Part 6: Functions. Available under the topic *Functions*.

Process:

The good news is that there is actually relatively little code for us to modify in order to change out direction commands from numbers to sentences.

Currently, our *getMovementDirection()* function returns an integer code that our main game loop checks to see where to move the player. If we keep using the same return values, then we won't need to modify any of the code in the game loop.

This means the player can type the command "move south", and if our input function still returns the value '2', then the rest of the game will continue to work without modification.

Because our game would be pretty boring if we could only move, we'll also add some other commands. In this tutorial we'll add 'look' and 'fight', but you could extend what we start here and add your own commands.

As long as our new commands don't return the value 2, 4, 6 or 8, then they won't be mistaken for movement directions.

And because we're adding these extra commands, we'll change the name of the *getMovementDirection()* function to *getCommand()*.

Adding New Commands:

Before we start on the changes to the input handling function, we need some new constant variables that our new commands ('look' and 'fight') will use.

Add the following code to the top of your *main.cpp* file (these should be global constants):

```
const char* EXTRA_OUTPUT_POS = "\x1b[25;6H";

const int PLAYER_INPUT_X = 30;
const int PLAYER_INPUT_Y = 23;

// input commands
const int SOUTH = 2;
const int EAST = 6;
const int WEST = 4;
const int NORTH = 8;

const int LOOK = 9;
const int FIGHT = 10;
```

We've also changed the value for *PLAYER_INPUT_Y* from 11 to 23. This will mean that the player will type their input below the map.

We've changed this because the player could potentially enter several lines of text before pressing enter. We want to be sure that the player doesn't start typing over the game map. It will also be easier for use to clear this area (so the user can input the next command), and gives us a bit more flexibility with how many lines we can output (so if the user types an invalid command, we can output an appropriate response).

Rather than jump in now and change our input function to allow for sentences, let's add the code to our main game loop that will handle the *Look* and *Fight* commands. Doing this first simply means we can test all our commands at once.

Find the switch statement in your game loop that handles the movement directions and update it as follows:

```
~~int direction = getMovementDirection();~~
int command = getCommand();
// 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 (command) {
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++;
    break;
case LOOK:
    drawPlayer(playerX, playerY);
    std::cout << EXTRA_OUTPUT_POS << RESET_COLOR << "You look around, but see
        nothing worth mentioning" << std::endl;
    std::cout << INDENT << "Press 'Enter' to continue.";
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    break;
case FIGHT:
    drawPlayer(playerX, playerY);
    std::cout << EXTRA_OUTPUT_POS << RESET_COLOR << "You could try to fight,
        but you don't have a weapon" << std::endl;
    std::cout << INDENT << "Press 'Enter' to continue.";
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    break;
default:
    // the direction was not valid,
    // do nothing, go back to the top of the loop and ask again
    drawPlayer(playerX, playerY);
    std::cout << EXTRA_OUTPUT_POS << RESET_COLOR << "You try, but you just
        can't do it." << std::endl;
    std::cout << INDENT << "Press 'Enter' to continue.";
    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    break;
}
```

We've change the name of the *getMovementDirection()* function, and added the case statements for *LOOK* and *FIGHT*.

In the future, we can look at moving the processing for these commands into their own function (once they have a bit more logic to them), but for now this is fine. We've also fleshed out the *default* statement to give the user an error message when they enter an invalid command.

Reading from the Command Line:

The rest of this tutorial covers the *getCommand()* function, which will read and interpret a line of text on the command line and output an integer corresponding to the command our game loop should execute.

When used to read text, *std::cin* will stop when it gets to a white space character.

Consider the following code:

```
char input[50] = "\0";
std::cin >> input;
```

If we type the following line of text in the console:

```
"eat a pineapple"
```

Then the value of *input* is actually the word "eat" and not the whole text "eat a pineapple".

Our basic approach will be to use a *while* loop to keep getting words from the command line until there is no more input left to read.

A naïve way to do this may be to write something like this:

```
char input[50] = "\0";
std::cin >> input;
while (input) {
    // check if the word in input is a command keyword
    ...

    std::cin >> input;
}
```

Our intent with this code is clearly to keep reading words until no more words are left in console's input buffer. But in reality we end up with an endless loop. Why?

If there is no more text to read from the console, *std::cin >> input* will simply wait until more text is entered. The player will see a blinking cursor, and if they enter more text then our loop will begin again.

To solve this problem, after we read a word from *cin* we need to 'peek' at the next character in the buffer (peeking lets us look at the character, but doesn't remove it from the buffer). If the character is a new-line character (i.e., the 'enter' key) or the end-of-file marker, then we want to exit the loop.

The updated code will look like this:

```
char input[50] = "\0";
std::cin >> input;
while (input) {
    // check if the word in input is a command keyword
    ...

    char next = std::cin.peek();
    if (next == '\n' || next == EOF)
        break;

    std::cin >> input;
}
```

Once you know this, it becomes fairly trivial to write a function that will 'read' each word and return the correct command code once a valid sentence has been entered.

Of course, there is an alternate method we could have used.

We could have read *all* of the text the player entered at once, using the *std::cin.getLine()* function. But this has a few disadvantages. First, no matter how large we make our input buffer (in the code extract above we allocated an array of 50 characters), we can never guarantee that our buffer is large enough to hold all the text the player entered. We'd need to dynamically resize our buffer according to what the player entered.

Also, we need to use more of the other string manipulation functions in the C Runtime Library to search and extract individual words from the larger string (something we get for free when using *std::cin*). While this isn't terribly difficult code to write, its none the less code we'd have to write (and test).

Challenge:

Before continuing, see if you can write the *getCommand()* function yourself. Use the *strcmp()* function to compare the words read from the command line to the command keywords like 'move', 'north', 'look', etc.

The complete code for the *getCommand()* function is on the next page.

```
int getCommand()
{
    // for now, we can't read commands longer than 50 characters
    char input[50] = "\0";

    // jump to the correct location
    std::cout << CSI << PLAYER_INPUT_Y << ";" << 0 << "H";
    // clear any existing text
    std::cout << CSI << "4M";

    std::cout << INDENT << "Enter a command.";
    // 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 >> input;
    std::cout << RESET_COLOR;

    bool bMove = false;
    while (input) {
        if (strcmp(input, "move") == 0) {
            bMove = true;
        }
        else if (bMove == true) {
            if (strcmp(input, "north") == 0)
                return NORTH;
            if (strcmp(input, "south") == 0)
                return SOUTH;
            if (strcmp(input, "east") == 0)
                return EAST;
            if (strcmp(input, "west") == 0)
                return WEST;
        }

        if (strcmp(input, "look") == 0) {
            return LOOK;
        }

        if (strcmp(input, "fight") == 0) {
            return FIGHT;
        }

        char next = std::cin.peek();
        if (next == '\n' || next == EOF)
            break;
        std::cin >> input;
    }

    return 0;
}
```

With the explanation above, it should be clear what we're doing in this function. Note that we've moved the curser for input and output to below the map.

Our sentence parsing is actually pretty stupid. It will simply look for the command keywords in the correct order, and completely ignore whatever else the player typed. You should be able to craft something a bit smarter for your own game.