Tutorial – Classes: Zorp part 9

Introduction:

In this tutorial we will refactor our existing code into classes. No new functionality will be added.

As we've seen over the last several tutorials, the complexity and scope of our project has been gradually increasing. It should be obvious that by continuing with a procedural programming approach (C style programming) our code will get more disorderly and difficult to work with as we add more functionality.

Using an Object Oriented Programming methodology is a good way to break our project into logical components that function together to create the game as a whole.

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*.
- Zorp Part 7: Character Arrays. Available under the topic *Arrays*.
- Zorp Part 8: Structures. Available under the topic *Structures and Classes*.

Classes:

We need to define a list of classes that are used in our game.

The most common way to do this is to describe what our game is (it's good to write this description down), and then to pick out all the nouns. The nouns will be the classes.

In our game, the player walks through a maze of rooms, battling enemies, and collecting food and treasure.

In this description, our nouns are:

- Game
- Player
- Maze
- Room
- Enemy

- Food
- Treasure

The next step is to decide which of these nouns need to be turned into classes, which ones we can combine into one class, and which ones we might want to represent through data or processing alone (that is, which ones might not need to be classes at all).

I've decided to make classes to represent the following:

- Game
- Player
- Room

Currently, we have room types (empty rooms, treasure rooms, enemy rooms, etc), so we don't need to represent things like enemies, treasure, and food with their own classes.

This might be a cool addition for the future. We could, for example, have an Enemy class which would allow us to have enemies that roam the map. But for now, let's limit ourselves to these three classes and concentrate only on refactoring our existing code base.

The *Player* class will hold the x and y coordinates of the player, and contain functions to draw the player and execute player related commands (specifically the movement commands).

The *Room* class will hold the x and y coordinate of the room, along with the type of room. Similar to the *Player* class, it will have functions for drawing the room and its description, and processing commands (look and fight).

The *Game* class is probably the most difficult to understand because it doesn't represent a "physical" object. It will be used to control the game logic. It will hold all our game objects (the rooms that make up the game map, and the player), and coordinate the communication between these objects.

The *Game* class is essentially a way to move all of our global functions and data out of the *main.cpp* file (and the *main()* function) and into a class so that we have a program that conforms to OOP convention.

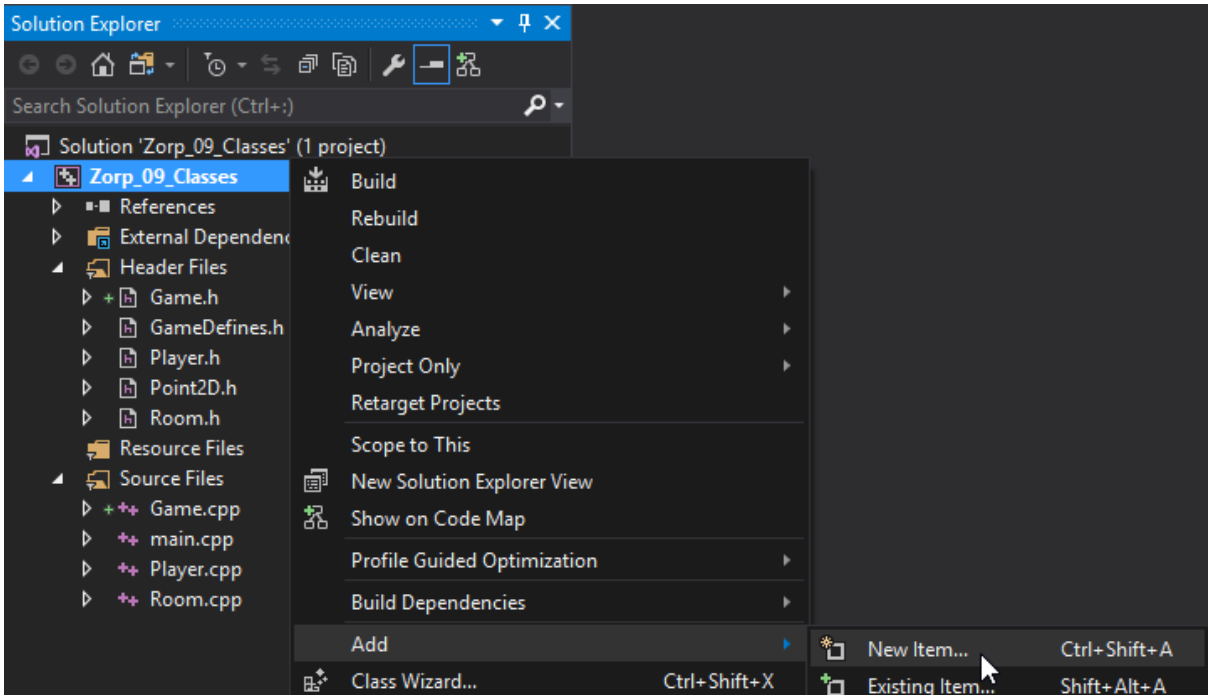
The Point2D Structure:

Before we begin creating our new classes, let's put the Point2D structure in its own header file.

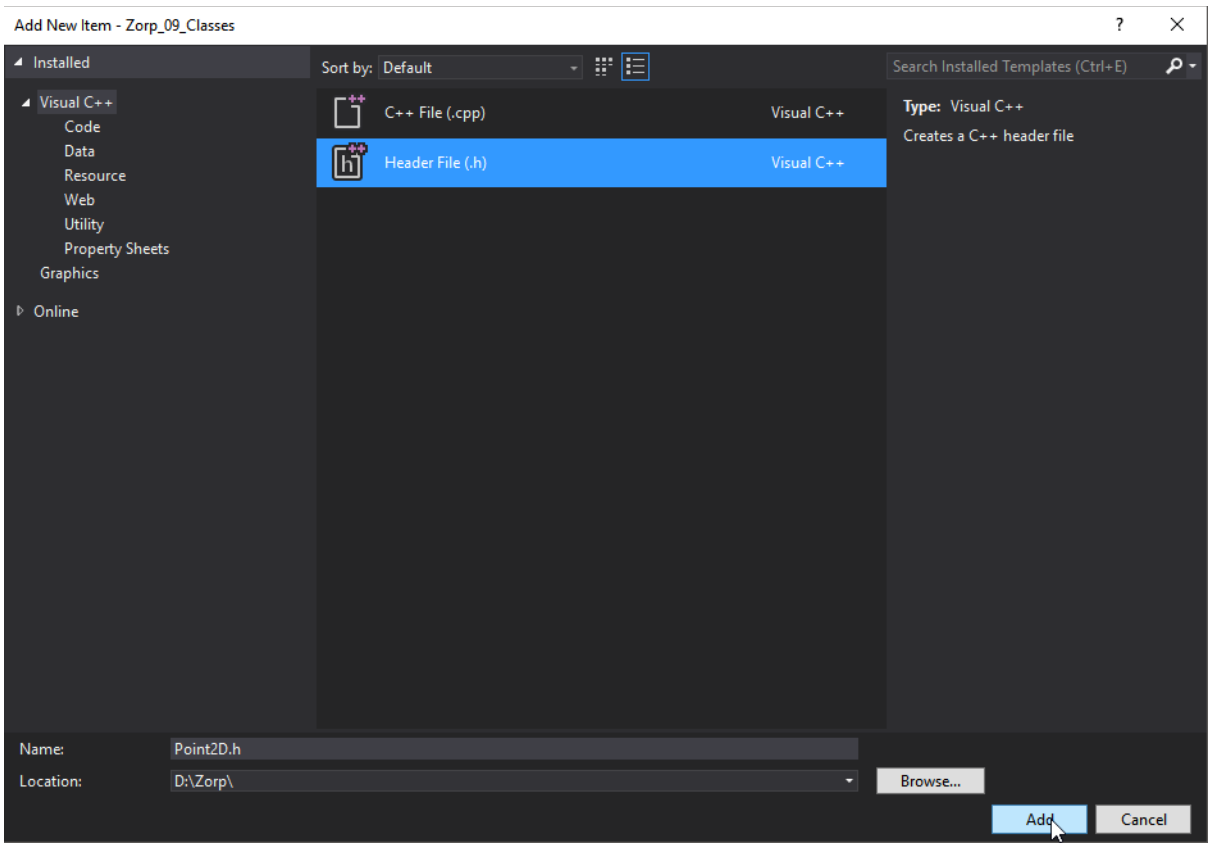
We want to do this so that the structure definition can be easily accessed throughout our program (necessary because we want to use Point2D objects to store the location of rooms, and the player)

First, we need to create a new header file called *Point2D.h*.

Right-click on the *Solution* name inside the *Solution Explorer* and select *Add -> New Item*



Create a new *header* file called *Point2D.h*



Press *Add* and your new file will be created, added to the project, and opened in the editor window.

Move the declaration of the *Point2D* structure from the *main.cpp* file to the *Point2D.h* file. Your new header file should contain the following text:

```
#pragma once

struct Point2D {
	int x;
	int y;
};
```

Global Constant Definitions:

We'll also want to do the same thing with our game constants. This is we can access things like the text colour control strings in all the classes that need to output text (hint, that's all of them).

Create a new header file called *GameDefines.h* and move all of the current global constant variables into this file.

The *GameDefines.h* file should look as follows:

```
#pragma once

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

const char* const TITLE = "\x1b[5;20H";
const char* const INDENT = "\x1b[5C";
const char* const YELLOW = "\x1b[93m";
const char* const MAGENTA = "\x1b[95m";
const char* const RED = "\x1b[91m";
const char* const BLUE = "\x1b[94m";
const char* const WHITE = "\x1b[97m";
const char* const GREEN = "\x1b[92m";
const char* const RESET_COLOR = "\x1b[0m";
const char* const EXTRA_OUTPUT_POS = "\x1b[25;6H";

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 = 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;
```

Take special care to update the definition of your constant strings to be *const char* const*.

We haven't learnt why this is necessary yet, and it's difficult to explain without the teaching material on pointers.

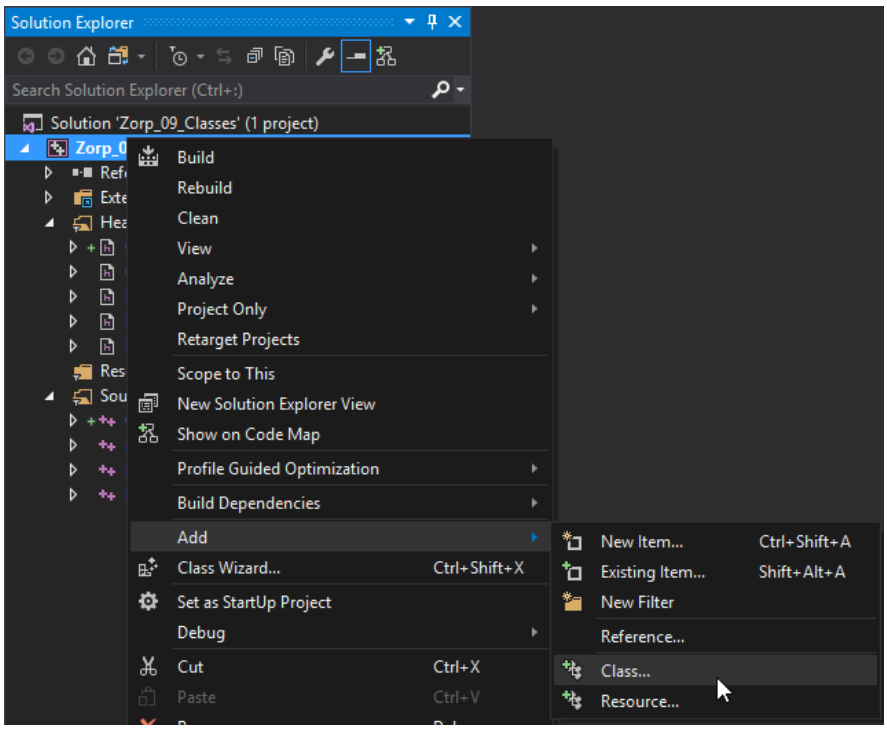
In short, for the strings variables, the variable name is actually an address in memory where the character string is stored. By using two *const* keywords, we're saying that both the address in memory *and* the value that is stored at that address are constant.

This will be discussed in greater detail in a future session. For now, the updated definition is necessary in order for our code to compile correctly.

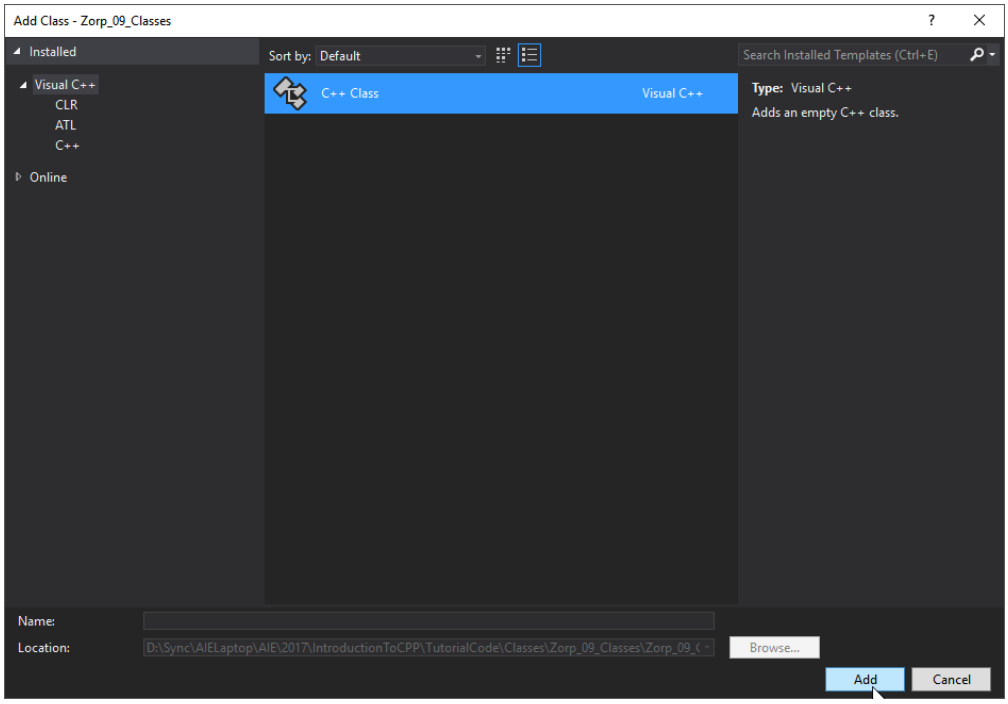
The Room Class:

When we refactor a game like this into classes, I think it helps by starting with the smallest or most easily defined class. Our Room class has the fewest dependencies (none) and is logically very simple (it just outputs text to the screen), so it's a good place to start.

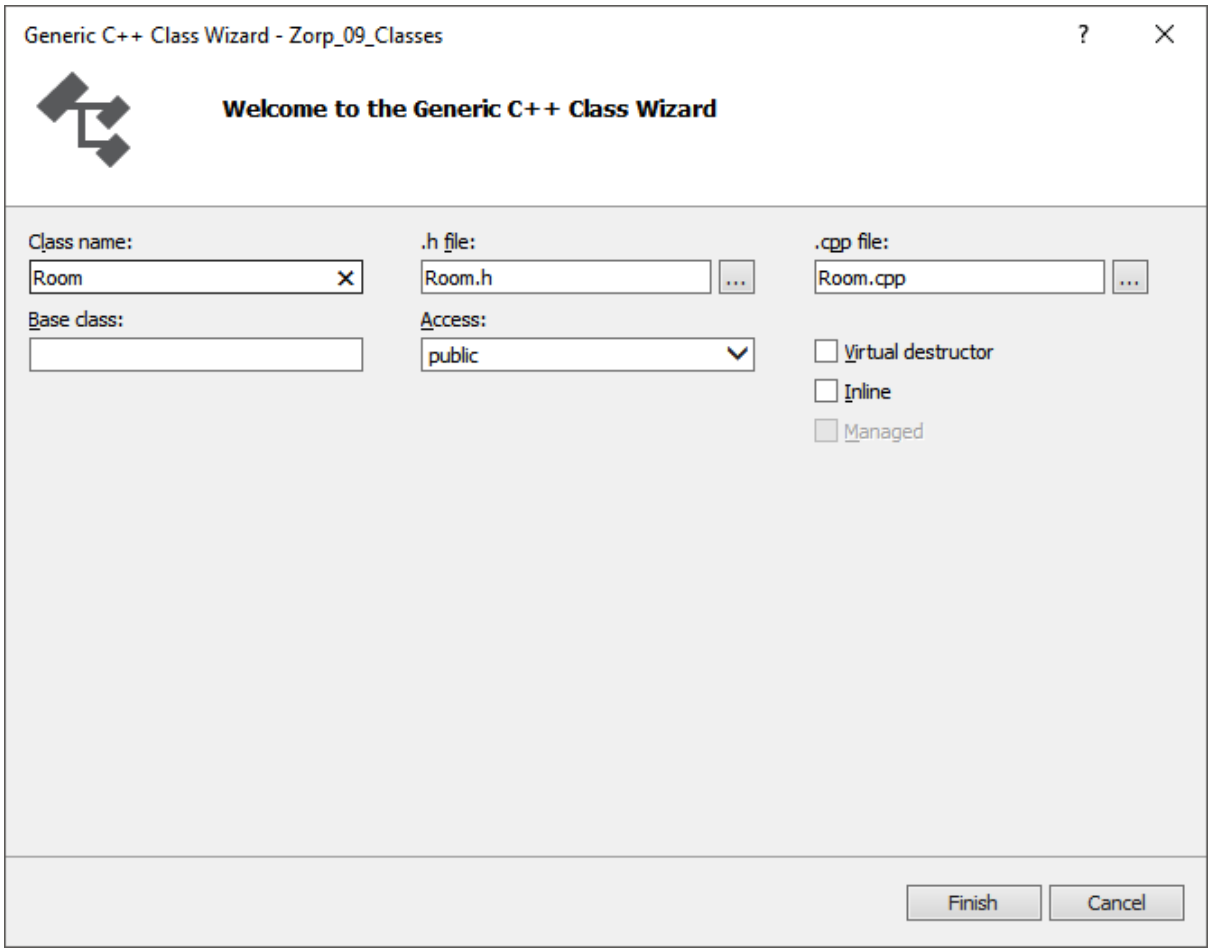
To make a new class in Visual Studio, right-click on the *Solution* name in the *Solution Explorer* and select *Add -> Class…*



Select *Class* then press *Add*



And fill in the information for your new class.



Type *Room* in the field for the *Class name*, and the .h and .cpp fields will be filled in automatically.

Press *Finish*, and the .h and .cpp files for your new class will be created. In the .h file you will see a very basic declaration of your new class containing only a constructor and destructor. The .cpp contains the implementation of these functions.

Once you've created the class, open *Room.h* and update the class declaration as follows:

```
#pragma once
#include "Point2D.h"

class Room
{
public:
	Room();
	~Room();

	void setPosition(Point2D position);
	void setType(int type);

	int getType();

	void draw();
	void drawDescription();

	bool executeCommand(int command);

private:
	Point2D m_mapPosition;
	int m_type;
};
```

The code to implement these functions comes directly from the current *main.cpp* file.

You should be able to write (or rather, copy and paste) these functions yourself. Unfortunately, you won't know if you made a mistake until you make and implement the remaining classes and try to run your game.

If you feel confident that you can implement the *Room* class yourself, then do that now and skip ahead to the next section.

If you need more guidance on writing this class, or want to check your implementation, continue reading.

Because the code we're using to implement the *Room* class's functions comes directly from code we've already written, we won't be explaining in detail what the code does.

The only new functionality here are the 'getters' and 'setters' that we've added to be able to access (get and set) the private member variables (m_mapPosition and m_type). We'll use the constructor to initialize these variables to a default value.

The following is the complete *Room.cpp* file:

```
#include "Room.h"
#include "GameDefines.h"
#include <iostream>

Room::Room() {
	m_type = EMPTY;
	m_mapPosition.x = 0;
	m_mapPosition.y = 0;
}

Room::~Room() {
}

void Room::setPosition(Point2D position) {
	m_mapPosition = position;
}

void Room::setType(int type) {
	m_type = type;
}

int Room::getType() {
	return m_type;
}

void Room::draw() {
	// find the console output position
	int outX = INDENT_X + (6 * m_mapPosition.x) + 1;
	int outY = MAP_Y + m_mapPosition.y;

	// jump to the correct location
	std::cout << CSI << outY << ";" << outX << "H";
	// draw the room
	switch (m_type) {
	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;
	}
}
```

(continued on next page)

```
void Room::drawDescription() {
	std::cout << RESET_COLOR; // reset draw colors
	std::cout << CSI << ROOM_DESC_Y << ";" << 0 << "H"; // jump to correct pos
	// Delete 4 lines and insert 4 empty lines
	std::cout << CSI << "4M" << CSI << "4L" << std::endl;
	// write description of current room
	switch (m_type) {
	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;
	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;
	}
}

bool Room::executeCommand(int command) {
	switch (command) {
	case LOOK:
		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();
		return true;
	case FIGHT:
		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();
		return true;
	default:
		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;
	}
	return false;
}
```

There are a few details worth highlighting.

The *draw()* function draws the room itself. Because we can't guarantee who is calling the function, we need to calculate the correct console output coordinates based on the room's map coordinates (stored as a member variable). This is the reason we store the room's coordinates as a variable as opposed to passing it in as a function argument.

The *executeCommand()* function will take a command (defined in *GameDefines.h*) and perform an action based on the command. If the function is called using an unrecognised command, it is handled in the same was as in the previous program (by writing an error message).

The Player Class:

The *Player* class, like the *Room* class, is also very simple. There is little logic, and all of the code comes from the existing *main.*cpp code.

Make sure as you copy the code across you adjust the variables to use the class's member variables.

Create a new class called *Player*, and update the header file as follows:

```
#pragma once
#include "Point2D.h"

class Player {
public:
	Player();
	Player(int x, int y);
	~Player();


	void setPosition(Point2D position);

	Point2D getPosition();

	void draw();

	bool executeCommand(int command);

private:
	Point2D m_mapPosition;
};
```

The implementation is quite simple. Note the *executeCommand* function, which only processes those commands that are related to player movement. If the function is called passing a command that is not a movement command, then the command is ignored and the function returns false to indicate the command was not 'consumed'.

The only thing we need to do is let the player process commands before passing any unprocessed command to the current room. This is because a movement command will be treated as an erroneous command by the room. This will become clear when we update the main game loop.

The *Player.cpp* file is as follows:

```
#include "Player.h"
#include "GameDefines.h"
#include <iostream>

Player::Player() {
	m_mapPosition.x = 0;
	m_mapPosition.y = 0;
}

Player::Player(int x, int y) {
	m_mapPosition.x = x;
	m_mapPosition.y = y;
}


Player::~Player() {
}


void Player::setPosition(Point2D position) {
	m_mapPosition = position;
}

Point2D Player::getPosition() {
	return m_mapPosition;
}

void Player::draw() {
	Point2D outPos = {
		INDENT_X + (6 * m_mapPosition.x) + 3,
		MAP_Y + m_mapPosition.y };

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

bool Player::executeCommand(int command) {
	switch (command) {
	case EAST:
		if (m_mapPosition.x < MAZE_WIDTH - 1)
			m_mapPosition.x++;
		return true;
	case WEST:
		if (m_mapPosition.x > 0)
			m_mapPosition.x--;
		return true;
	case NORTH:
		if (m_mapPosition.y > 0)
			m_mapPosition.y--;
		return true;
	case SOUTH:
		if (m_mapPosition.y < MAZE_HEIGHT - 1)
			m_mapPosition.y++;
		return true;
	}
	return false;
}
```

The Game Class:

The game class will control our whole game.

We'll store the rooms 2D array (which we'll rename to *map*), along with the player variable.

All the remaining functions in our *main.cpp* will be added as member functions to the *Game* class.

Create a new class, called *Game*, and update its declaration as follows:

```
#pragma once

#include "GameDefines.h"
#include "Room.h"
#include "Player.h"

class Game{
public:
	Game();
	~Game();

public:
	bool startup();
	void update();
	void draw();

	bool isGameOver();

private:
	bool enableVirtualTerminal();
	void initializeMap();

	void drawWelcomeMessage();
	void drawMap();
	void drawValidDirections();

	int getCommand();

private:
	bool m_gameOver;
	Room m_map[MAZE_HEIGHT][MAZE_WIDTH];
	Player m_player;
};
```

The *startup()*, *update()*, and *draw()* functions will be called by the *main()* function. All other functions in this class will only be called by the class itself, and so should be *private*.

update() and *draw()* will be called each time the game loop is executed. This mimics what you would find in traditional graphical games, where we would be calling these functions every frame (typically 60 times per second).

It's very important to only put code that modifies variables or objects (i.e., updates our game objects) in the *update()* function, and code that outputs or draws the game objects in the *draw()* function.

While it's not a terribly important distinction to make in this program (or any text-based program for that matter), it will be important in the future games you make. So it's good to get into the habit now of separating your games into 'update logic' and 'draw logic'.

Let's take the class functions in order, starting with the constructor, destructor and *startup()* functions:

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

Game::Game() {
	m_gameOver = false;
}

Game::~Game() {
}

bool Game::startup() {
	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 false;
	}

	initializeMap();

	m_player.setPosition(Point2D{ 0,0 });

	drawWelcomeMessage();

	return true;
}
```

The constructor simply sets the *m_gameOver* variable to false. We won't initialize any other variables here, because that will be done during the *startup()* function. (We could have left the *m_gameOver* variable to be initialize in *startup()* too. This really comes down to personal preference, as long as it's clear that the value *is* being initialized somewhere).

The *startup()* function will enable the virtual terminal mode, fill our game map with random rooms, set the position of the player to the start room, then draw the welcome message. You'll notice that this is everything that happened before the game loop started in our old program.

startup() will return *false* if there was a problem launching the game (i.e., if the virtual terminal mode couldn't be set). The *main()* function will need to check for this when the game starts.

The next functions are those that are called by the *startup()* function (*enableVirtualTerminal(), initializeMap(),* and *drawWelcomeMessage()*). I've also thrown in the definition of the *isGameOver()* function.

```
bool Game::isGameOver() {
	return m_gameOver;
}

bool Game::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;
}

void Game::initializeMap() {
	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)
				m_map[y][x].setType(type);
			else
				m_map[y][x].setType(EMPTY);
			m_map[y][x].setPosition(Point2D{ x, y });
		}
	}
	// set the entrance and exit of the maze
	m_map[0][0].setType(ENTRANCE);
	m_map[MAZE_HEIGHT - 1][MAZE_WIDTH - 1].setType(EXIT);
}

void Game::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;
}
```

This code should be familiar from the previous tutorials.

For the *drawMap()* function, we're simply looping through the 2D array and calling the *draw()* function for each room.

You should have noticed that during the *initializeMap()* function we explicitly set the position of each room by calling the room's *setPosition()* function. This means that we don't need to pass the x and y indices to the *draw()* function because the room already knows it's location in the map.

```
void Game::drawMap()
{
	Point2D position = { 0, 0 };

	// reset draw colors
	std::cout << RESET_COLOR;
	for (position.y = 0; position.y < MAZE_HEIGHT; position.y++)
	{
		for (position.x = 0; position.x < MAZE_WIDTH; position.x++) {
			m_map[position.y][position.x].draw();
		}
		std::cout << std::endl;
	}
}

void Game::drawValidDirections()
{
	Point2D position = m_player.getPosition();

	// 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 " <<
		((position.x > 0) ? "west, " : "") <<
		((position.x < MAZE_WIDTH - 1) ? "east, " : "") <<
		((position.y > 0) ? "north, " : "") <<
		((position.y < MAZE_HEIGHT - 1) ? "south, " : "") << std::endl;
}
```

The last function before we tackle the *Game*'s *update()* and *draw()* functions is the *getCommand()* function.

This function doesn't actually reference either the player or the map, so we can simply copy and paste the existing *getCommand()* function into the *Game* class.

```
int Game::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;
}
```

That leaves us with the *update()* and *draw()* function.

We need to take the logic that happens inside the game loop in the existing *main()* function and break it down into the two tasks ‘updating’ and ‘drawing’.

This should be reasonably easy for you to identify.

The updating will be getting the command to execute from the user (via a call to *getCommand)*, and then executing that command (by passing the command to the player and the current room in a call to the *executeCommand()* function).

The drawing that we need to do each time through the game loop is drawing the directions the player can move in, drawing the current room description, drawing the map, and finally drawing the player.

In previous tutorials we only redrew the previous room the player was in (so that we could remove the player's marker from the room after they left the room). This same logic will be a little challenging to achieve now that we are separating our game into 'updating' and 'drawing'. To achieve this we'd need to store the player's previous location. Instead of doing this, we'll simply redraw the whole map each time we execute the game loop.

The *update()* and *draw()* functions are as follows:

```
void Game::update() {
	Point2D playerPos = m_player.getPosition();

	if (m_map[playerPos.y][playerPos.x].getType() == EXIT) {
		m_gameOver = true;
		return;
	}

	int command = getCommand();

	if (m_player.executeCommand(command))
		return;

	m_map[playerPos.y][playerPos.x].executeCommand(command);
}

void Game::draw() {
	Point2D playerPos = m_player.getPosition();

	// list the directions the player can take
	drawValidDirections();

	// draw the description of the current room
	m_map[playerPos.y][playerPos.x].drawDescription();

	// redraw the map
	drawMap();
	// draw the player on the map
	m_player.draw();
}
```

That's the whole *Game* class. It's important to understand how we arrived at this code from our previous procedural program, so make sure you review the classes we've written if you need to.

The *main()* Function:

The final task is to update our *main()* function.

Because we've moved the actual game logic into the *Game* class, the *main()* function has been simplified greatly.

All that is necessary for the *main()* function to do is create an instance of the *Game* class, call its's *startup()* function, and then run a game loop that calls the *Game's update()* and *draw()* function every loop until the game ends.

The updated *main.cpp* file is below.

Of special note is the fact we need to call the *Game's draw()* function before *update()*. This is because the update call will block the execution of the game while it waits for the players input. In a normal (graphical) game, we'd typically call the *update()* function first.

```
#include <iostream>
#include "Game.h"

void main() {
	Game game;

	if (game.startup() == false)
		return;

	// game loop
	while (!game.isGameOver())
	{
		game.draw();

		game.update();
	} // 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();
}
```

Compile and execute your program. If all went as expected, your game should function exactly the same as it did before.

If you ran into trouble and cannot figure out your errors, the complete source code from this tutorial can be downloaded from the *Resources* page for this subject on Canvas.