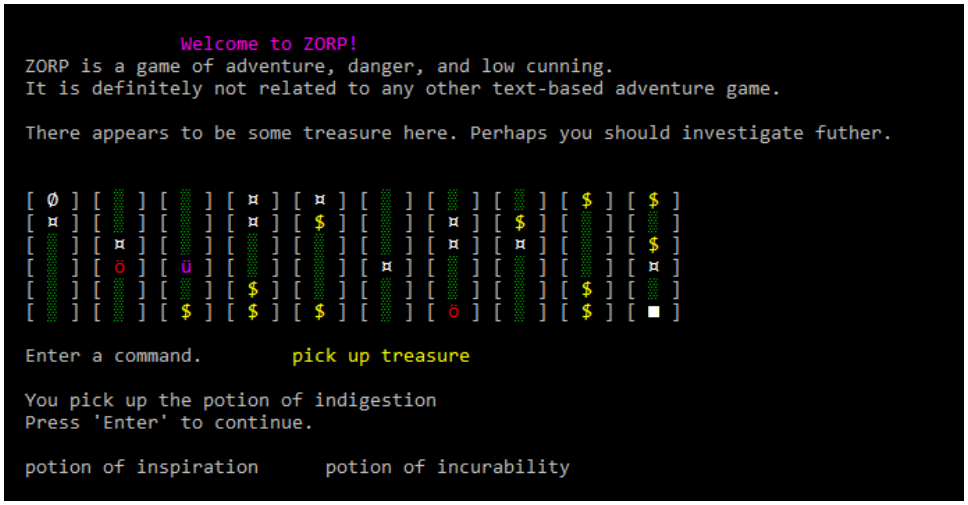
Tutorial – STL Vector: Zorp part 11

Introduction:

In this tutorial we are going to expand the *Player* class by adding an 'inventory'. The inventory will track power-ups that the player has picked up. Power-ups will be found in the treasure rooms.

In this tutorial we will only add the basic functionality to add a power-up to the player's inventory, and print out a list of power-ups the player currently has. We aren't intending to finish this system in this tutorial as this will require learning from future sessions.

At the end of this tutorial the player will be able to pick up a random item in a treasure room. The item will be added to the *Player* class's internal list (implemented as an *std::vector*), and displayed to the screen. For now, the power-up items will have no effect on gameplay.



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*.
- Zorp Part 9: Classes. Available under the topic *Structures and Classes*.
- Zorp Part 10: Constructors and Destructors. Available under the topic *Structures and Classes*.

The Powerup Class:

We want our player to collect power-ups as they progress through the map.

Each power up should have a name, and a series of player attributes it affects. In our game, a power-up can affect a player's health (HP), attack ability (AT), or defence ability (DF). These modifiers will be real numbers where the value 1.0 means the item has no effect on that attribute, a value less than 1 will decrease the attribute, and a value greater than 1 will increase the attribute.

Each time the player's *update()* function is executed, the modifiers are multiplied by the original attribute value to give the current total for that attribute.

To encapsulate that information in a class, we'll define the *Powerup* class as follows:

```
#pragma once

class Powerup
{
public:
	Powerup(const char name[30], float health, float attack, float defence);
	~Powerup();

	char* getName();
	float getHealthMultiplier();
	float getAttackMultiplier();
	float getDefenceMultiplier();

private:
	char m_name[30];

	float m_healthMultiplier;
	float m_attackMultiplier;
	float m_defenceMultiplier;
};
```

You should be able to write the implementation of the functions and constructor yourself. There is no logic in this class, the functions will simply store and retrieve the member variables.

Updating the Map:

Each of our treasure rooms should spawn a random powerup.

We're going to have three different kinds of powerup: a potion to modify HP, a sword to modify AT, and a shield to modify DF.

We want to randomly allocate one of these treasure types to each treasure room, so we're going to need a few more constant variable definitions.

We'll also take this opportunity to modify the position of our output so that we have space to print the player's inventory.

Update the *GameDefines.h* file 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[21;6H";
const char* const INVENTORY_OUTPUT_POS = "\x1b[24;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 TREASURE_HP = 6;
const int TREASURE_AT = 7;
const int TREASURE_DF = 8;

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 = 12;
const int PLAYER_INPUT_X = 30;
const int PLAYER_INPUT_Y = 19;

// 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;
const int PICKUP = 11;
```

We've also added one more command ('pick up') to allow us to pick up these powerup items.

To turn our standard treasure rooms into rooms with either potions, swords or shields, update the following functions in the *Game.cpp* file:

```
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)
			{
				if (type == TREASURE)
					type = rand() % 3 + TREASURE_HP;
				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);
}
```

Every time a *treasure* room is randomly placed on the map, we'll do one more calculation to randomly determine which of the three types of treasure rooms it should be.

Here, the modulus operation (rand() % 3) will ensure the random number generated is between 0 and 3. By adding the value *TREASURE_HP* to this random value, the value of *type* will be one of the three new treasure room values (HP, AT, or DF).

We'll also need to add the command that allows the player to pick up an item while in a treasure room. As with the other commands, we'll keep it simple and only look for the works 'pick' and 'up'.

```
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";
	// insert 4 blank lines to ensure the inventory output remains correct
	std::cout << CSI << "4L";

	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;
        bool bPickup = 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;
                }
                if (strcmp(input, "pick") == 0) {
                        bPickup = true;
                }
                else if (bPickup == true) {
                        if (strcmp(input, "up") == 0)
                                return PICKUP;
                }

                char next = std::cin.peek();
                if (next == '\n' || next == EOF)
                        break;
                std::cin >> input;
        }
        return 0;
}
```

If we were to run the game now, even though the new treasure rooms are being created they wouldn't be drawing to the screen,

We need to update the *Room* class to ensure the output is correct for our new room types.

We should also update the description the player sees when they enter the treasure rooms, so they know there are items they can pick up.

Update the following functions in the *Room.cpp* file:

```
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_HP:
	case TREASURE_AT:
	case TREASURE_DF:
		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;
	}
}
```

Remember, this switch statement works because the flow of execution will pass straight through to the next case statement if no *break* is encountered. This is not possible in some languages.

```
void Room::drawDescription()
{
...

	switch (m_type) {
	...
	case TREASURE_HP:
	case TREASURE_AT:
	case TREASURE_DF:
		std::cout << INDENT << "There appears to be some treasure here.
				Perhaps you should investigate futher." << std::endl;
		break;
	...
	}
}
```

The update to the *drawDescription()* function is similar to that for the *draw()* function. We remove the case for the old *TREASURE* room type, and replace it with case statements for the three new room types. The rest of the code has not been shown as it remains the same.

```
bool Room::executeCommand(int command) {
	std::cout << EXTRA_OUTPUT_POS;
	switch (command) {
	case LOOK:
		if (m_type == TREASURE_HP || m_type == TREASURE_AT || m_type ==
			TREASURE_DF ) {
			std::cout << EXTRA_OUTPUT_POS << RESET_COLOR <<
				"There is some treasure here. It looks small enough to
				pick up." << std::endl;
		}
		else {
			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:
		// the direction was not valid,
		// do nothing, go back to the top of the loop and ask again
		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;
}
```

The last function to update, *executeCommand()*, will process the '*look*' command. This will inform the player that they can pick up a treasure item, but only if they are currently in a treasure room.

Notice that we have not added the '*pick up*' command here. Because this command modifies the *Player* class object (by adding an item to the player's inventory), we'll need to process this command inside the *Player*'s *executeCommand()* function.

The Player's Inventory:

We're now ready to extend the *Player* class by adding an inventory of powerups. We'll start by defining an *std::vector* inside the *Player* class definition, which we'll use to store the powerups the player picks up.

Update the *Player.h* as follows:

```
#pragma once

#include "Point2D.h"
#include "Powerup.h"
#include <vector>

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


	void setPosition(Point2D position);

	Point2D getPosition();

	void draw();

	bool executeCommand(int command, int roomType);

private:
	bool pickup(int roomType);

private:
	Point2D m_mapPosition;

	std::vector<Powerup> m_powerups;

	int m_healthPoints;
	int m_attackPoints;
	int m_defendPoints;
};
```

We've also added a new argument to the *executeCommand()* function. This is because of the new '*pick up*' command. We need to know what room the player is in when they execute that command, as it will only work from within a treasure room.

Although we won't use the health, attack, or defend points just yet, you should go ahead and update the constructors.

```
Player::Player() : m_mapPosition{ 0, 0 }, m_healthPoints{100}, m_attackPoints{20},
m_defendPoints{20}
{}

Player::Player(int x, int y) : m_mapPosition{ x, y }, m_healthPoints{ 100 },
m_attackPoints{ 20 }, m_defendPoints{ 20 }
{}
```

First, we want to fill in the body of the *pickup()* function. This new function will be called from the *executeCommand()* function, and simply encapsulates the processing that occurs when the 'pick up' command is typed in by the player.

We've separated this processing into its own function so that the *executeCommand()* function remains small and legible.

Here is the code for the *picup()* function:

```
bool Player::pickup(int roomType)
{
	static const char itemNames[15][30] = {
		"indifference", "invisibility", "invulnerability", "incontinence",
		"improbability", "impatience", "indecision", "inspiration",
		"independence", "incurability", "integration", "invocation",
		"inferno", "indigestion", "inoculation"
	};

	int item = rand() % 15;
	char name[30] = "";

	switch (roomType) {
		case TREASURE_HP:
			strcpy(name, "potion of ");
			break;
		case TREASURE_AT:
			strcpy(name, "sword of ");
			break;
		case TREASURE_DF:
			strcpy(name, "shield of ");
			break;
		default:
			return false;
	}

		// append the item name to the string
	strncat(name, itemNames[item], 30);
	std::cout << EXTRA_OUTPUT_POS << RESET_COLOR <<
		"You pick up the " << name << std::endl;
	m_powerups.push_back(Powerup(name, 1, 1, 1.1f));

	std::cout << INDENT << "Press 'Enter' to continue.";
	std::cin.clear();
	std::cin.ignore(std::cin.rdbuf()->in_avail());
	std::cin.get();
	return true;
}
```

The first thing this function does is declare a static constant array of strings. These are the names of the random items in the treasure rooms.

One of these names will be chosen at random, and combined with the string "potion of", "sword of", or "shield of" (depending on the room type) to create the item name (for example, "potion of incurability").

We then output the name of the item the player collected before creating a new *Powerup* class object and pushing it to the back of the *m_powerups* vector.

Now we can update the *executeCommand()* function, ensuring that the *pickup()* function in called whenever the *PICKUP* command identifier is received.

```
bool Player::executeCommand(int command, int roomType) {
	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;
	case PICKUP:
		return pickup(roomType);
	}
	return false;
}
```

Remember that we now need to pass the *roomType* into this function because the *pickup* command is only valid inside the treasure rooms.

Now that we have some objects for the player to pick up and place in their inventory, we should have a way to display the contents of that inventory.

In the player's *draw()* function we want to step through each element in the *m_powerups* vector and print the name of each *Powerup* object to the console.

To do this, we'll use an iterator. To simplify the code, we'll use the *auto* keyword when declaring the iterator variable, although we could have chosen to write the complete type (`std::vector<Powerup>::iterator`) instead.

If you don't completely understand how the *auto* keyword works, it is better to specify the complete type yourself.

```
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;

	std::cout << INVENTORY_OUTPUT_POS;
	for (auto it = m_powerups.begin(); it < m_powerups.end(); it++) {
		std::cout << (*it).getName() << "\t";
	}
}
```

The '\t' escape character specifies a tab. We've used it to add a bit more whitespace between the output items.

Before we can execute our updated program, we just need to make sure we are passing the room type to the *Player*'s *executeCommand()* function from the *Game*'s *update()* function:

```
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,
		m_map[playerPos.y][playerPos.x].getType()))
		return;

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

That's it! Compile and execute your game now. You should be able to pick up items from the treasure rooms and have them display in your player's inventory.

If you're observant, you may have noticed that we didn't add processing to prevent us from picking up more than one object from each treasure room (i.e., we haven't notified the room that the object has been taken).

The solution to this problem involves the use of pointers, and will be explained in a future tutorial.