Tutorial – Pointers to Memory: Zorp part 13

Introduction:

In this tutorial we'll see how we can create pointers to objects, and use these pointers as input to function calls in order to pass these objects around our program.

We'll update our program to pass a *Room* object pointer to the *Player* class so it can extract the information it needs when processing the *PICKUP* command.

Because passing the *Room* object as a pointer in this way gives the *Player* class the flexibility to modify the room, once a player picks up the powerup from a treasure room we'll set that room's type to *EMPTY*.

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*.
- Zorp Part 11: STL Vector. Available under the topic *Standard Template Library*.
- Zorp Part 12: STL Algorithms. Available under the topic *Standard Template Library*.

Pointer Arguments:

Rather than pass an integer indicating the type of room the player is in, it would be preferable to pass a *pointer* to the *Room* object itself. This greatly extends the flexibility of the program and gives us the ability to implement features that would otherwise have been difficult or cumbersome to implement.

For example, by passing a *Room* pointer into the *Player's executeCommand()* function, we can query or even modify the room during the execution of a command.

This will be very useful for us when processing the *PICKUP* command.

Currently there is no way for us to limit how many times a player executes the *PICKUP* command inside a treasure room. You may have noticed that you can keep typing "pick up" while in a treasure room, and the program will keep adding a random powerup to your inventory.

Previously we had no way to prevent this because the only information the *Player*'s *executeCommand()* function had access to was the command and the room type (both integers).

Any solution that we might have come up with previously would have been overly complex, and possibly let to other problems.

But, by passing in a *pointer* to the *Room* object, we'll now have the ability to modify the properties of the room while executing the *PIKUP* command.

First, we need to update the *Player* class's header, changing the arguments for the *executeCommand()* and *pickup()* functions from the *roomType* integer to a *Room pointer*.

```
class Room;

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

    void setPosition(Point2D position);

    Point2D getPosition();

    void draw();

    bool executeCommand(int command, Room* pRoom);

private:
    bool pickup(Room* room);

private:
    Point2D m_mapPosition;

    std::vector<Powerup> m_powerups;

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

The changes to the *pickup()* and *executeCommand()* functions are on the next page.

See if you can implement this functionality yourself before continuing.

We've forward declared the *Room* class in the updated *Player* class header above. This means you don't need to include *Room.h* in the header (to avoid any possible cyclic dependencies). But you will need to include *Room.h* inside *Player.cpp*.

```
bool Player::pickup(Room* room) {
    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 (room->getType()) {
        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::sort (m_powerups.begin(), m_powerups.end(), Powerup::compare);
    // set the room to EMPTY
    room->setType(EMPTY);

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

bool Player::executeCommand(int command, Room* room) {
    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(room);
    }
    return false;
}
```

From Object to Pointer:

The last step to perform is to call the updated *executeCommand()* function from the *Game*'s *update()* function.

The *Room*s are stored in the *Game* class as objects (in a 2D array), but the *executeCommand()* function now requires a *Room pointer*.

To convert an object variable to a pointer, we use the ampersand (&). As described in the lecture, we can think of this as the 'memory address' operator. It will give us the address in memory where a variable's value is stored. 'Memory address' is another name for a *pointer*.

Update the *Game* class's *update()* function 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, &m_map[playerPos.y][playerPos.x]))
        return;

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

Make sure you include the & character at the front of the *m_map* variable.

Compile and execute your program.

It should function the same as before, except now after you collect an item from a treasure room the room will be set to empty.