BIT 2400 – Intermediate Programming

# Tutorial 4 – Classes

**Important:**

- Do not lose the code you write today. Save it somewhere. We will keep adding to this code for the next tutorials.
- If you don't complete the lab (there's a lot), I will post solutions to each lab so you can use that the following tutorial.
- You can grab the code from last lab to start. It's more gratifying if you use your own code but if you had problems, you can use the provided code. Use Tutorial 2 code to make your GameGrid class.

This tutorial will focus on classes and their constructors and destructors.

## Game Overview

Now the fun part….or "fun" part. In the following weeks, we will incrementally make a very basic zombie survival game. No one expects a perfect game that will make you a millionaire but the task is open-ended enough that beginners can just do the basic task and have a game while more experienced programmers can show off their creativity and skill. Here are the instructions for the following weeks:

We are making a text based zombie survival "game" involving a player with weapons and different types of undead. This is what we are building towards

### Game Mechanics

- Each player would have a movement speed, and weapon types. You could also have hit points.
- You have a starting position and all undead attempt to travel to you each turn.
- You input the next destination position each turn and move towards that location based on your movement speed XOR you type another key to use one of your weapons (XOR means one or the other).
- You could have users have all weapons or have them pick up weapons when they walk over certain positions.
- Each turn the positions of each undead and the player is printed out as a grid. Typing a new position or using a weapon starts the next turn.
- The super class of zombie is undead (zombie28 is another subclass). Vampire and ghost also extend (are subclasses of) undead.
    - Each undead class has a different weakness so some weapons don't affect it. For example, "stake" only hurts vampires. Chainsaw only affects zombie….you would think they would "affect" a vampire but not according to the movies.
    - Undead enemies have different speeds as well

    - If the undead touches a player (without the player correctly attacking it), the player dies.
    - If the player uses a weapon that affects THAT undead type and the enemy is within range of the weapon, the undead dies. More than one enemy can die that turn.
    - The player may or may not know what type of undead each creature is (but may infer it by movement speed).
    - Optional: If an enemy dies, some undead types might run away from the player for a couple of rounds. Zombies won't run away.
- You win if you defeat all undead or lose if they touch you.

Seems simple enough right? Well so far no one has made a final game (in previous terms I attempted to cover this in 2 labs). The real point of this game is to teach programming basics but I still hope we can get a real game working.

**Important: Do not just delete your code week after week.** I want you to add to your code and eventually make this game. Make sure you save your code somewhere safe.

**Each week I will release solutions to the tutorial. If you miss a week or you couldn't finish the tutorial, feel free to use these solutions as starter code for the next week.**

**I will show a video of the game in action in the early tutorial videos.**

## A. Create the Zombie class

We are about to make a command-line based game called Zombie Madness and the first step is to create the enemies: the zombies.

A zombie walks around a field (it has an X and Y position) and can go slow or fast. It can also be freshly dead (it can still eat people), or very dead (it can no longer eat anyone but can move). Zombies also attempt to attack the player and if they reach the player, they attack.

- Create the class Zombie.
- Create the **protected** attributes for this class from the information above (4 attributes) plus 2 more attributes: the destination x position and the destination y position. This is where the zombie is trying to go (slowly)
- Create 4 functions:
    - **setDestination** takes two integer parameters (xPos and yPos) and makes that the zombie's desired location.
    - Move the zombie towards its target or goal position: the method should be called **takeTurn** and it has no parameters. The function returns an integer indicating the amount of damage the zombie does to the player (thus most times you expect a 0).

- Print the zombie's position and some information about the zombie. This function should be called **printInfo**. printInfo doesn't return anything or take any parameters but uses cout statements to output information to the screen.
- The function **hit** will not take any parameters but instead returns the amount of damage the zombie does (let's say it returns the value 3 for "3 hit points" from the player. If the zombie is very dead, the value returned is 0). Notice this value is then returned from takeTurn.
- You should probably also make getter methods for various attributes as well. You can choose what attributes other classes need to have access to. This is for a game

### Moving

- Moving the zombie should be limited to how fast it travels.
- Let's say a typical zombie goes 3 "steps" per turn where each step is over 1 position in the x OR y direction. Thus moving from (3, 3) to (8, 3) would mean the zombie is at (6, 3) after a turn.
- Moving from (4, 3) to (1, 2) could move the zombie to (4, 2) but not to (2,2) until the following turn.
- A fast zombie (we will make a fast zombie class next week) may be able to run 6 positions from (4, 3) to (1, 2) in a single turn but not from (1,1) to (8, 9).
- The **takeTurn()** function updates the zombie location if it is not already in the desired location. Thus, in the fast zombie case above takeTurn moves it from (1,1) to (4,4) and then to (7,7) the next turn. One more turn and it stops at (8, 9)

Create a .h file that contains the class definition (including function prototypes, and a .cpp file that contains the function definitions.

Make sure you test that the zombie class works. Give it a destination and call takeTurn a couple times, printing the zombie information between each turn.

## C. Create your main file

You can use the following code for your.cpp file to test your code. HOWEVER, you will need to make any functions you have not yet implemented but that are being called in the code below. This is just so you test your code sooner.

```
#include  "zombie.h"
#include  <iostream>
using namespace std;


void main()
{
	cout << "1. Zombie z1" << endl;
	Zombie z1;
	z1.setDestination(3, 4);
	z1.printInfo();

	cout << endl << "2. Zombie z2" << endl;
```

```
        Zombie* z2 = new Zombie(8, 8);
        z2->setDestination(5, 5);

        cout << endl << "3. Zombie z3" << endl;
        Zombie z3(22, 2, shotgun);
        z3.setFast(true);
        z3.setDestination(2, 2);

        // Update the positions one additional time
        z1.takeTurn();
        z2->takeTurn();
        z3.takeTurn();

        delete z2;
}
```

It should produce the following output:

```
1. Zombie z1
constructor: zombie()
The zombie is now at position (0,0) going to (3,4)

2. Zombie z2
constructor: zombie(int, int)
The zombie is now at position (8,8) going to (5,5)

3. Zombie z3
constructor: weapon(int)
constructor: zombie(int, int, int)
The zombie is now at position (22,2) going to (2,2)

The zombie is now at position (2,1).
The zombie is now at position (6,7).
The zombie is now at position (16,2).
destructor: ~zombie()
destructor: ~zombie()
destructor: ~weapon()
destructor: ~zombie()
Press any key to continue . . .
```

Observe the preceding output and try to understand which call corresponds to which zombie (z1, z2, or z3).

## D. Create the Weapon class

What's the point of zombies without weapons? Each weapon has a type (int or enum type), range (int) and a force (int)

- Create the class Weapon.
- Create the attributes for this class from the information above (3 attributes).
- Create the appropriate functions:
    - Constructors & Destructors

        - What should the defaults be?
        - Make a constructor that takes the weapon type as a parameter (and knows what the range and force should be based on that)
    - **Get/Set functions** for every attribute. The "getter" function associated with an attribute returns that attribute, and the "setter" function of an attribute takes a new value as argument and updates the attribute with this new value.
        - For example: int getRange() and void setRange(int newRange) could be a getter and setter for the range attribute. One sets how far the weapon reaches or shoots and the other gets that value for other objects to see.
- For each constructor, destructor, and function, write a **temporary** output statement with the name of the function & its arguments. For instance, this would be the output from the weapon constructor by default.
    - `cout << "constructor: weapon()" << endl;`

You should delete these statements after we finish this lab.

Create a .h file that contains the class definition, and a .cpp file that contains the functions.

Now in your main method, create a weapon and a zombie and then call various functions for each to test your code.

## Types of weapons

I made an enumerated data type called weapon type that I stuck in my Weapon.h

```
typedef enum
{
    shotgun = 1,
    butterKnife = 2,
    katana = 3
} weaponType;
```

This means you can write shotgun to represent 1 instead of using the value 1. A switch statement might look like this:

```
switch (t)
    {
    case shotgun:
        range = 3;
        force = 3;
        break;
    case katana:
        range = 1;
```

You do not have to follow this approach but it's a useful tactic. You could also define SHOTGUN_ID to be 1, KATANA_ID to be 2, etc.

## E. Modify the Zombie class

Modify the zombie class to include a weapon variable. Wait. Zombies have weapons? They do for now (you'll look at this more next week). This weapon is a dynamic object (pointer). You should use "new" and "delete" to create and remove it. You should only create it when necessary.

- Add the attribute
- Create the appropriate functions (get/set)
- Create / Update the default constructors & destructors of zombie, and create 2 other constructors, one that takes as argument the xy position, and the other that takes the xy position and the type of weapon.
- For each constructor and destructor, and an output statement with the name of the function & its arguments. For instance, this would be the output from the zombie constructor by default.
    - `cout << "constructor: zombie()" << endl;`
- Add a cout line in every constructor & destructor to observe the call order.
- Continue to test your code with these new changes.

## F. Migrate the game controls (you can delay doing this until L07 if you want)

Take code from tutorial 2 and modify it to create a GameGrid object. The GameGrid object is the entirety of the game we are creating. It would handle user input, displaying the grid to users, and updating positions.

- Remember that you will not need so many parameters once these functions are in a class; the player and enemy positions are stored as member variables.
- This is a great task if you want to make sure you understand objects and information hiding. GameGrid has to know about the various objects (like Movers, the player, and enemies) and communicate with them to move enemies, make the player attack, etc.
- I created a Mover class that each enemy class and the player class have as member objects. You tell your mover object to move to position (5, 6) and give it a speed and the object will adjust the position an appropriate amount each turn until the player or enemy reaches (5,6). This mover class really helped simplify my code and made things reusable.
- Given the length of this lab I will create this class and you can use or modify the solution. This was just my way of implementing things. There are many other approaches you could take but if you are looking at my code you'll see how I'm getting access to member variables, how I'm passing information around, and how I break down a complex problem into smaller independent chunks of code. You should be able to use this code in subsequent assignments.

Also note that I “refactored” the tutorial 3 code to remove hard coded values and make the code more modular.

## Submit your work

Make sure you submit your work on Brightspace. Feel free to use the posted solution next tutorial (it’s a long lab after all) but you should definitely try to do as much of this work as you can.