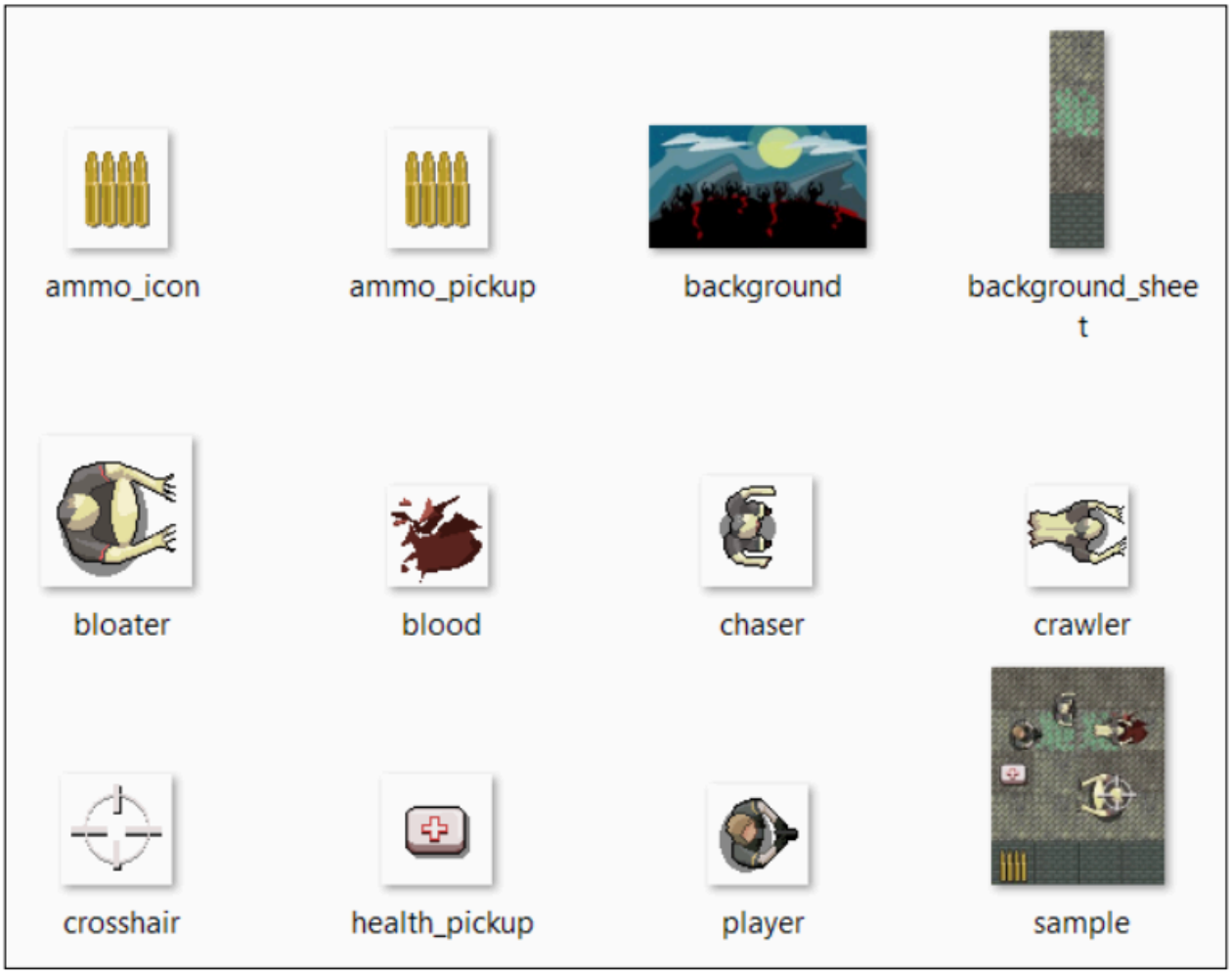
Exploring the assets

The graphical assets make up the parts of the scene of the Zombie Arena game.



***** Create createBackground.cpp**

createBackground.cpp

```
#include "ZombieArena.h"

int createBackground(VertexArray& rVA, IntRect arena)
{
	const int TILE_SIZE = 50;
	const int TILE_TYPES = 3;
	const int VERTS_IN_QUAD = 4;

	int worldWidth = arena.width / TILE_SIZE;
	int worldHeight = arena.height / TILE_SIZE;


	rVA.setPrimitiveType(Quads);

	rVA.resize(worldWidth * worldHeight * VERTS_IN_QUAD);

	int currentVertex = 0;


	for (int w = 0; w < worldWidth; w++)
	{
		for (int h = 0; h < worldHeight; h++)
		{
			rVA[currentVertex + 0].position = Vector2f(w * TILE_SIZE, h * 
TILE_SIZE);
			rVA[currentVertex + 1].position = Vector2f((w * TILE_SIZE) + 
TILE_SIZE, h * TILE_SIZE);
			rVA[currentVertex + 2].position = Vector2f((w * TILE_SIZE) + 
TILE_SIZE, (h * TILE_SIZE) + TILE_SIZE);
			rVA[currentVertex + 3].position = Vector2f((w * TILE_SIZE), (h * 
TILE_SIZE) + TILE_SIZE);
```

```
                        if (h == 0 || h == worldHeight - 1 || w == 0 || w == worldWidth - 1)
                        {
                                // Use the wall texture
                                rVA[currentVertex + 0].texCoords = Vector2f(0, 0 +
TILE_TYPES * TILE_SIZE);
                                rVA[currentVertex + 1].texCoords = Vector2f(TILE_SIZE, 0
+ TILE_TYPES * TILE_SIZE);
                                rVA[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
TILE_SIZE + TILE_TYPES * TILE_SIZE);
                                rVA[currentVertex + 3].texCoords = Vector2f(0, TILE_SIZE
+ TILE_TYPES * TILE_SIZE);
                        }

                        else
                        {
                                // Use a random floor texture
                                srand((int)time(0) + h * w - h);

                                int mOrG = (rand() % TILE_TYPES);
                                int verticalOffset = mOrG * TILE_SIZE;
                                rVA[currentVertex + 0].texCoords = Vector2f(0, 0 +
verticalOffset);
                                rVA[currentVertex + 1].texCoords = Vector2f(TILE_SIZE, 0
+ verticalOffset);
                                rVA[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
TILE_SIZE + verticalOffset);
                                rVA[currentVertex + 3].texCoords = Vector2f(0, TILE_SIZE
+ verticalOffset);

                        }

                        currentVertex = currentVertex + VERTS_IN_QUAD;
                }
        }

        return TILE_SIZE;
```

}

Main Function

ZombieArena.cpp

```
#include <SFML/Graphics.hpp>
#include "Player.h"
#include "createBackground.cpp"
using namespace sf;

int main()
{
    Vector2f resolution;
    resolution.x = VideoMode::getDesktopMode().width;
    resolution.y = VideoMode::getDesktopMode().height;

    RenderWindow window(VideoMode(resolution.x, resolution.y),
                        "Zombie Arena", Style::Fullscreen);

    View mainView(FloatRect(0, 0, resolution.x, resolution.y));

    IntRect arena(0, 0, 1000, 1000);

    VertexArray background = createBackground(arena);

    Texture backgroundTexture;
    backgroundTexture.loadFromFile("graphics/background_sheet.png");

    Player player;
    player.spawn(resolution);

    Clock clock;

    while (window.isOpen())
```

```
    {
        if (Keyboard::isKeyPressed(Keyboard::Escape)) window.close();
        if (Keyboard::isKeyPressed(Keyboard::W)) player.moveUp(); else player.stopUp();
        if (Keyboard::isKeyPressed(Keyboard::S)) player.moveDown(); else
player.stopDown();
        if (Keyboard::isKeyPressed(Keyboard::A)) player.moveLeft(); else
player.stopLeft();
        if (Keyboard::isKeyPressed(Keyboard::D)) player.moveRight(); else
player.stopRight();

        Time dt = clock.restart();
        float dtSeconds = dt.asSeconds();

        Vector2i mousePosition = Mouse::getPosition(window);
        player.update(dtSeconds, mousePosition);

        mainView.setCenter(player.getPosition());
        window.setView(mainView);


       window.clear(Color::Black);
        window.draw(background, &backgroundTexture);
        window.draw(player.getSprite());
        window.display();
    }

    return 0;
}
```

Important Questions

Q1.

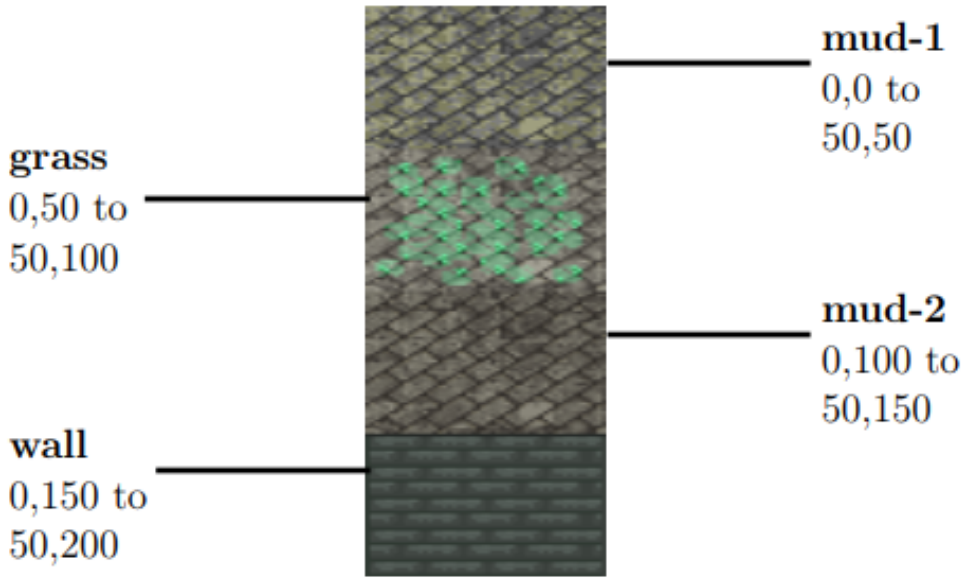
Write SFML-C++ code snippet to declare a vertex array with Quads type primitive and size of the vertex array 10 × 10 × 4.

(10 × 10 × 4: For a 10×10 tile grid, total vertices = 100 quads × 4 = 400)

```
#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{

   VertexArray background;

   background.setPrimitiveType(Quads);

   background.resize(10 * 10 * 4);

   return 0;
}
```

Q2.

Assume that ***background sheet.png*** sprite sheet is given to you. Write SFML-C++ code snippet to draw 3 tiles (mud-1, grass, mud-2) onto the screen. you are free to decide the position of each vertex in the current quad and texture coordinates will be selected as per the given sprite sheet.



Tile	***Texture Coor***
mud 1	(0, 0) to (50, 50)
gras s	(0, 50) to (50, 100)

mud (0, 100) to (50, 150)
2

Draw these 3 tiles horizontally on screen

---

mud-1 at (0, 0)
grass at (50, 0)
mud-2 at (100, 0)

Code 1:

```
#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{
   VideoMode VM(400, 400)
   RenderWindow window(VM, "Tiled Background");

   Texture texture;
   texture.loadFromFile("background_sheet.png"));

   const int TILE_SIZE = 50;
   const int TILE_TYPES = 3;  // mud-1, grass, mud-2
   const int GRID_WIDTH = 4;
   const int GRID_HEIGHT = 4;

//
```

Total tiles = 4 * 4 = 16
Each tile has 4 vertices = 16 * 4 = 64
I want to draw 16 tiles using 64 vertices total.

```
    VertexArray background(Quads, GRID_WIDTH * GRID_HEIGHT * 4);

    int currentVertex = 0;

    for (int w = 0; w < GRID_WIDTH; ++w)
    {
        for (int h = 0; h < GRID_HEIGHT; ++h)
        {
            background[currentVertex + 0].position = Vector2f(w * TILE_SIZE,
h * TILE_SIZE);
            background[currentVertex + 1].position = Vector2f((w + 1) *
TILE_SIZE, h * TILE_SIZE);
            background[currentVertex + 2].position = Vector2f((w + 1) *
TILE_SIZE, (h + 1) * TILE_SIZE);
            background[currentVertex + 3].position = Vector2f(w * TILE_SIZE,
(h + 1) * TILE_SIZE);


            int tileIndex = (w + h) % TILE_TYPES;
            int verticalOffset = tileIndex * TILE_SIZE;


            background[currentVertex + 0].texCoords = Vector2f(0,
verticalOffset);
            background[currentVertex + 1].texCoords = Vector2f(TILE_SIZE,
verticalOffset);
```

```
            background[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
verticalOffset + TILE_SIZE);
            background[currentVertex + 3].texCoords = Vector2f(0, verticalOffset
+ TILE_SIZE);

            currentVertex += 4;
        }
    }

    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event))
            if (event.type == Event::Closed)
                window.close();

        window.clear();
        RenderStates states;
        states.texture = &texture;
        window.draw(background, states);
        window.display();
    }

    return 0;
}
```

RenderStates object - it's a structure that holds extra information for rendering, such as:

A **transform**
A **shader**
A **blend mode**
A **texture**

states.texture = &texture;
When we draw this vertex array, use this texture image (background_sheet.png).

Q3.

Design a function with the given prototype ***displayBackground(VertexArray& rVA, IntRect arena)***; to draw the background over the window as per the following structure using the above sprite sheet. (***Grass across center row***)

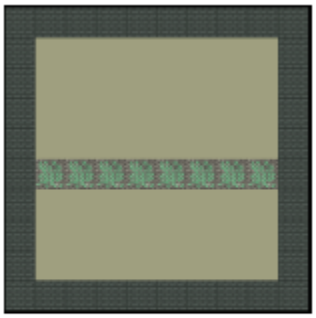


Figure 3: Arena Background

Tile	*Texture Coor*
mud 1	(0, 0) to (50, 50)
gras s	(0, 50) to (50, 100)
mud 2	(0, 100) to (50, 150)
Wall	(0,150) to (50,200)

Code 1

```
void displayBackground(VertexArray& rVA, IntRect arena)
{
  const int TILE_SIZE = 50;
  const int VERTS_IN_QUAD = 4;

  int worldWidth = arena.width / TILE_SIZE;
  int worldHeight = arena.height / TILE_SIZE;

  rVA.setPrimitiveType(Quads);
  rVA.resize(worldWidth * worldHeight * VERTS_IN_QUAD);
```

```
int currentVertex = 0;


for (int w = 0; w < worldWidth; w++) {

    for (int h = 0; h < worldHeight; h++) {

            rVA[currentVertex + 0].position = Vector2f(w * TILE_SIZE, h * TILE_SIZE);

        rVA[currentVertex + 1].position = Vector2f((w + 1) * TILE_SIZE, h * TILE_SIZE);

         rVA[currentVertex + 2].position = Vector2f((w + 1) * TILE_SIZE, (h + 1) * TILE_SIZE);

        rVA[currentVertex + 3].position = Vector2f(w * TILE_SIZE, (h + 1) * TILE_SIZE);


        Vector2f texTL;


        // Wall

        if (w == 0 || w == worldWidth - 1 || h == 0 || h == worldHeight - 1)

       {

           texTL = Vector2f(0, 150);

        }

        // Grass strip across center row
```

```
        else if (h == worldHeight / 2)
      {
          texTL = Vector2f(0, 50);
        }
        // Otherwise mud 1
        else {
          texTL = Vector2f(0, 0);
        }

        // Apply texture coordinates
        rVA[currentVertex + 0].texCoords = texTL;
          rVA[currentVertex + 1].texCoords = texTL + Vector2f(TILE_SIZE, 0);
          rVA[currentVertex + 2].texCoords = texTL + Vector2f(TILE_SIZE, TILE_SIZE);
                  rVA[currentVertex + 3].texCoords = texTL + Vector2f(0, TILE_SIZE);

        currentVertex += VERTS_IN_QUAD;
      }
    }
```

}

Main Function :

```
#include <SFML/Graphics.hpp>
using namespace sf;
void displayBackground(VertexArray& rVA, IntRect arena);
int main()
{
        VideoMode vm(500, 500)
      RenderWindow window(vm, "Arena Background");

   IntRect arena(0, 0, 500, 500);
  VertexArray background;
  displayBackground(background, arena);

   Texture texture;
  texture.loadFromFile("background_sheet.png"));

  while (window.isOpen())
  {
```

```
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
                window.close();
        }

        window.clear();
        RenderStates states;
        states.texture = &texture;
        window.draw(background, states);
        window.display();
    }

    return 0;
}
```

Q4.

Design a function with the given prototype **displayBackground(VertexArray& rVA, IntRect arena)**; to draw the background over the window as per the following structure using the above sprite sheet. (***Vertical or Horizontal grass cross in center***)

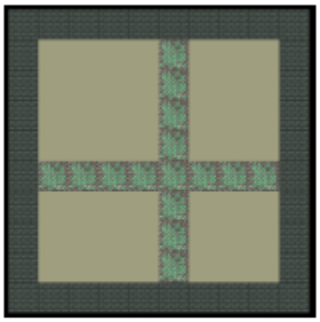


Figure 4: Arena Background

```
#include <SFML/Graphics.hpp>

using namespace sf;


void displayBackground(VertexArray& rVA, IntRect arena)

{

    const int TILE_SIZE = 50;

    const int VERTS_IN_QUAD = 4;


    int worldWidth = arena.width / TILE_SIZE;

    int worldHeight = arena.height / TILE_SIZE;
```

```
    rVA.setPrimitiveType(Quads);
    rVA.resize(worldWidth * worldHeight * VERTS_IN_QUAD);

    int currentVertex = 0;

    for (int w = 0; w < worldWidth; w++)
    {
        for (int h = 0; h < worldHeight; h++)
        {
            int x = w * TILE_SIZE;
            int y = h * TILE_SIZE;

            rVA[currentVertex + 0].position = Vector2f(x, y);
            rVA[currentVertex + 1].position = Vector2f(x + TILE_SIZE, y);
                rVA[currentVertex + 2].position = Vector2f(x + TILE_SIZE, y +
TILE_SIZE);
            rVA[currentVertex + 3].position = Vector2f(x, y + TILE_SIZE);

            Vector2f texCoords;
```

```
// Border wall
if (h == 0 || h == worldHeight - 1 || w == 0 || w == worldWidth - 1)
{
    texCoords = Vector2f(0, 150); // wall (0,150) to (50,200)
}
// Vertical or Horizontal grass cross in center
else if (w == worldWidth / 2 || h == worldHeight / 2)
{
    texCoords = Vector2f(0, 50); // grass (0,50) to (50,100)
}
// either mud-1 or mud-2
else
{
    texCoords = Vector2f(0, 0); // mud-1 (0,0) to (50,50)
}

// texture coordinates
rVA[currentVertex + 0].texCoords = texCoords;
                rVA[currentVertex + 1].texCoords = texCoords + Vector2f(TILE_SIZE, 0);
```

```
                rVA[currentVertex + 2].texCoords = texCoords +
Vector2f(TILE_SIZE, TILE_SIZE);

                rVA[currentVertex + 3].texCoords = texCoords + Vector2f(0,
TILE_SIZE);


            currentVertex += VERTS_IN_QUAD;

        }

    }

}
```

Q5.

The following ***Figure:1 shows event handling by polling***. Develop a code snippet to instantiate an object of Event type (Event is a SFML class type) to poll for the system events. Additionally, include a loop with the condition ***window.pollEvent(...)*** to keep looping each frame until there are no events to process and ***display the appropriate message when there is a state change.***

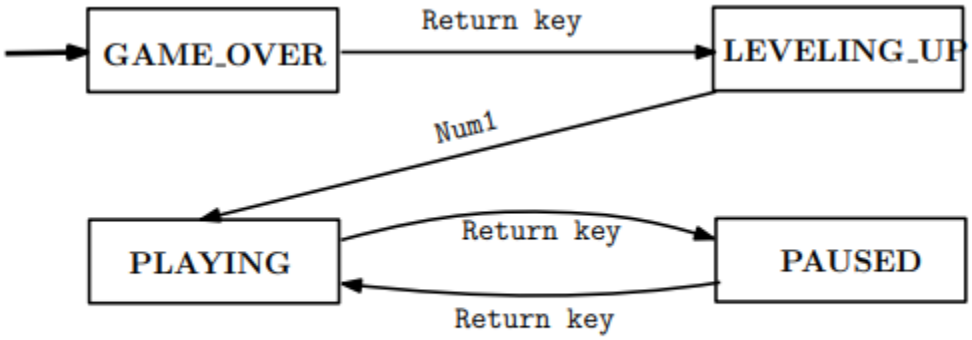


Figure 1: Game state transition for handling events by polling

Game State Flow

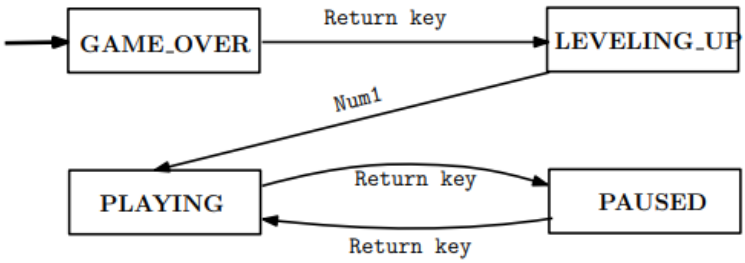
The game has the following states

GAME_OVER

LEVELING_UP

PLAYING

PAUSED



Each state represents what the game is currently doing. We can transition between states using keys.

GAME_OVER → LEVELING_UP

- ☐ Press Enter
- ☐ We are starting a new game from game over
- ☐ Game logic switches to LEVELING_UP

LEVELING_UP → PLAYING

- ☐ Press Num1, Num2, ..., Num6
- ☐ We are selecting a power-up to begin the level
- ☐ Game transitions to PLAYING state
- ☐ Player is spawned
- ☐ Zombies are created
- ☐ Arena is initialized
- ☐ Game begins

PLAYING ↔ PAUSED

- ☐ Press Enter
- ☐ We are pausing or resuming the game.
- ☐ From PLAYING → PAUSED: Game halts
- ☐ From PAUSED → PLAYING: Game resumes and clock is reset to sync gameplay.

Code

```
#include <SFML/Graphics.hpp>

#include <iostream>

using namespace sf;

using namespace std;


enum class State { GAME_OVER, LEVELING_UP, PLAYING, PAUSED };


int main()

{

  VideoMode vm(960, 540);

   RenderWindow window(vm, "Game State");

   State state = State::GAME_OVER;
```

```
Clock clock;
while (window.isOpen())
{
    Event event;
    while (window.pollEvent(event))
    {
        if (event.type == Event::Closed)
            window.close();
        if (event.type == Event::KeyPressed)
        {
            if (event.key.code == Keyboard::Return)
            {
                switch (state)
                {
                    case State::GAME_OVER:
                        state = State::LEVELING_UP;
                        cout << "State: LEVELING_UP\n";
                        break;
```

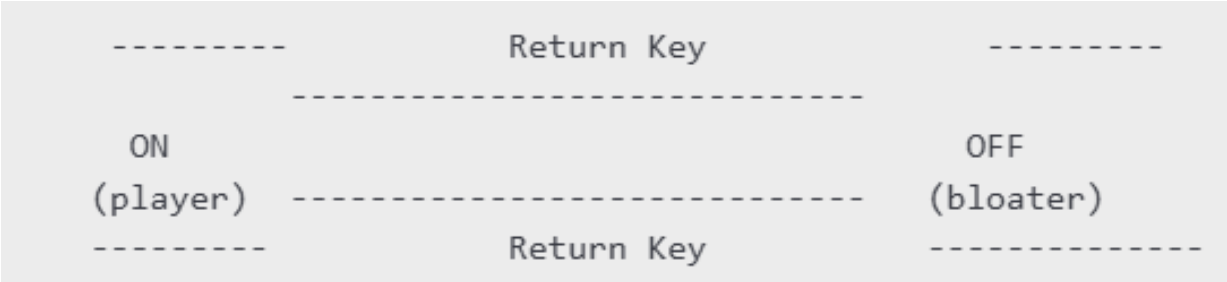
```
        case State::LEVELING_UP:
            state = State::PLAYING;
            cout << "State: PLAYING\n";
            break;

        case State::PLAYING:
            state = State::PAUSED;
            cout << "State: PAUSED\n";
            break;

        case State::PAUSED:
            state = State::PLAYING;
            cout << "State: PLAYING\n";
            clock.restart();
            break;
    }
} // end if return

if (state == State::LEVELING_UP)
{
```

```
                    if (event.key.code >= Keyboard::Num1 && event.key.code <=
Keyboard::Num6)
                {
                    state = State::PLAYING;
                    cout << "Power selected. State: PLAYING\n";
                }
            }
        } // end if keypressed
    } // end pollevent while loop

    if (Keyboard::isKeyPressed(Keyboard::Escape))
        window.close();

     window.clear();
     window.display();
  }  // Loop
  return 0;
}
```

Q6.

Assume ***ON and OFF*** are two states in a game with sprites ***player.png and bloater.png***. Initially the game is in ON state and the sprite, player, is drawn onto the game window. ***Game state can be changed with the key pressed Return.***

Construct a program to ***draw player sprite in ON state*** and ***bloater sprite in OFF state***. window.clear(Color::Red); may be used to change in default background color.



- ☐ player.png is drawn in the ON state.
- ☐ bloater.png is drawn in the OFF state.
- ☐ Pressing Return switches between the states.
- ☐ The background is cleared with Color::Red.

```
#include <SFML/Graphics.hpp>

Using namespace sf;


enum class GameState { ON, OFF };
```

```
int main()
{
   VideoMode vm(960, 540);
   RenderWindow window(vm, "State Switching");

    Texture playerTexture;
    playerTexture.loadFromFile("player.png")
    Sprite playerSprite;
    playerSprite.setTexture(playerTexture);
    playerSprite.setPosition(200, 300);

   Texture bloaterTexture;
  bloaterTexture.loadFromFile("bloater.png")
   Sprite bloaterSprite;
    bloaterSprite.setTexture(bloaterTexture);
    bloaterSprite.setPosition(200, 300);

    GameState state = GameState::ON;
```

```
    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event))
        {
              if (event.type == Event::Closed)
               window.close();


                if (event.type == Event::KeyPressed && event.key.code ==
Keyboard::Return)
            {
               if (state == GameState::ON)
                  state = GameState::OFF;
               else
                  state = GameState::ON;
            }
        } //end pollevent

        window.clear(Color::Red);
```

```
        if (state == GameState::ON)
        {
            window.draw(playerSprite);
        }
        else
        {
            window.draw(bloaterSprite);
        }
        window.display();
    }
    return 0;
}
```

Q7.

Rewrite the ***spawn public member function*** for the ***Player class*** to ***spawn 5 players*** as shown in the below Figure-2. Also state the SFML statements to call the ***spawn(...) function*** and ***window.draw(...)*** for the player.

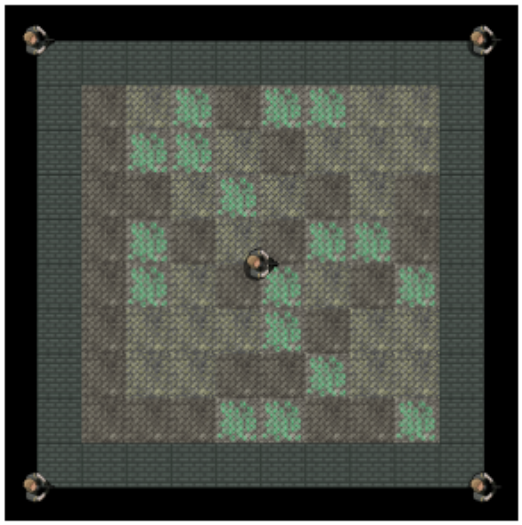


Figure 2: Player spawn at different position of the arena

Modified Code

Player.h

```
#include <SFML/Graphics.hpp>

using namespace sf;


class Player {

private:
```

```
    Sprite m_Sprite;
    Texture m_Texture;
    Vector2f m_Position;
public:
    Player();
    void spawn(Vector2f position);
    Sprite getSprite();
};
```

Player.cpp

```
#include "Player.h"
Player::Player() {
    m_Texture.loadFromFile("graphics/player.png");
    m_Sprite.setTexture(m_Texture);
    m_Sprite.setOrigin(25, 25);
}
void Player::spawn(Vector2f position) {
    m_Position = position;
    m_Sprite.setPosition(m_Position);
}
```

```
Sprite Player::getSprite() {
    return m_Sprite;
}
```

ZombieArena.cpp

```
#include <SFML/Graphics.hpp>
#include "Player.h"
int main() {
VideoMode vm (960, 540)
RenderWindow window(vm, "Spawn 5 Player");

    Player players[5];
    Vector2f spawnPoints[5] = {
        Vector2f(50, 50),      // Top-left
        Vector2f(450, 50),     // Top-right
        Vector2f(250, 250),    // Center
        Vector2f(50, 450),     // Bottom-left
        Vector2f(450, 450)     // Bottom-right
    };

    for (int i = 0; i < 5; ++i) {
```

```
        players[i].spawn(spawnPoints[i]);
    }

    while (window.isOpen()) {
        Event event;
        while (window.pollEvent(event)) {
            if (event.type == Event::Closed)
                window.close();
        }

        window.clear(Color::Red);

        for (int i = 0; i < 5; ++i) {
            window.draw(players[i].getSprite());
        }
        window.display();
    }
    return 0;
}
```

Q8.

Design a public member function, ***void spawn(float startX, float startY, int type, int seed)***, for the Zombie class to draw the ***3 kinds of zombies over the arena*** as shown in the below Figure-3. Also write THREE function call statements to invoke the spawn(...) function with three independent objects of that class Zombie.

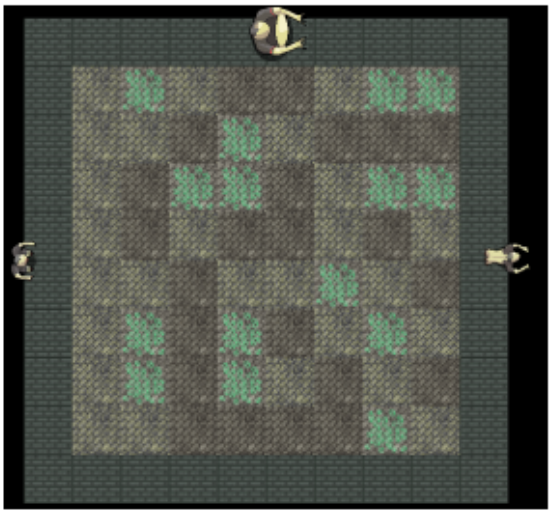


Figure 3: Zombies spawning over the arena wall

Zombie.h

```
#include <SFML/Graphics.hpp>

using namespace sf;


class Zombie {

private:
```

```
    Sprite m_Sprite;
    Texture m_Texture;
    Vector2f m_Position;

public:
    void spawn(float startX, float startY, int type, int seed);
    Sprite getSprite();
};
```

Zombie.cpp

```
#include "Zombie.h"
#include <cstdlib>
#include <ctime>

void Zombie::spawn(float startX, float startY, int type, int seed) {
    switch (type) {
        case 0:
            m_Texture.loadFromFile("graphics/bloater.png");
            break;
        case 1:
```

```
            m_Texture.loadFromFile("graphics/chaser.png");
            break;
        case 2:
            m_Texture.loadFromFile("graphics/crawler.png");
            break;
        default:
            m_Texture.loadFromFile("graphics/bloater.png");
            break;
    }

    m_Sprite.setTexture(m_Texture);
    m_Sprite.setOrigin(25, 25);
    m_Position.x = startX;
    m_Position.y = startY;
    m_Sprite.setPosition(m_Position);

    srand(seed);
}
```

```
Sprite Zombie::getSprite() {
   return m_Sprite;
}
```

main.cpp

```
#include "Zombie.h"

Zombie zombie1, zombie2, zombie3;

// We are placing the first zombie at position (250, 10)
// Type of zombie (0 = Bloater, 1 = Chaser, 2 = Crawler)

zombie1.spawn(250, 10, 0, 1);    // Bloater at top
zombie2.spawn(10, 250, 1, 2);    // Chaser at left
zombie3.spawn(490, 250, 2, 3);   // Crawler at right

window.draw(zombie1.getSprite());
window.draw(zombie2.getSprite());
window.draw(zombie3.getSprite());
```

Q9.

Redesign the above code snippet to ***use an array of objects rather than 3 individual objects of the Zombie class.*** Don't write the **spawn(...)** function again.

```
#include "Zombie.h"

using namespace std;

int main()

{

    VideoMode vm(960, 540);

    RenderWindow  window(vm, "Zombie Array");


   const int NUM_ZOMBIES = 3;

   Zombie  zombies[NUM_ZOMBIES];


   // Call spawn for each zombie in the array


   zombies[0].spawn(100, 100, 0, 1);

   zombies[1].spawn(200, 200, 1, 2);

   zombies[2].spawn(300, 300, 2, 3);
```

```
    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
                window.close();
        }

        window.clear();
        // Draw each zombie
        for (int i = 0; i < NUM_ZOMBIES; i++)
        {
            window.draw(zombies[i].getSprite());
        }
        window.display();
    }
    return 0;
}
```

Q10.

Redesign the above code snippet to ***use dynamic allocation of objects (using new) rather than array of objects of the Zombie class***. Don't write the spawn(...) function again.

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

int main()
{
    Zombie* zombies[3];

[Dynamically allocates memory for each Zombie object]
    zombies[0] = new Zombie;
    zombies[0]->spawn(100, 100, 0, 1);

    zombies[1] = new Zombie;
    zombies[1]->spawn(200, 100, 1, 2);

    zombies[2] = new Zombie;
```

```
    zombies[2]->spawn(300, 100, 2, 3);


    window.draw(zombies[0]->getSprite());
    window.draw(zombies[1]->getSprite());
    window.draw(zombies[2]->getSprite());


    for (int i = 0; i < 3; i++)
{
        delete zombies[i];
    }
    return 0;
}
```

Q11.

Find the output of the following code snippet;

```
int main()
{
int num=10;
int&  rnum = num;
```

```
int  &r1num = rnum;

rnum = 100;

cout<<rnum<<" "<<num<<" "<<r1num<<endl;

return 0;

}
```

References in C++

- ☐ A reference is an alias for another variable.
- ☐ Syntax: **int& ref = original;**
- ☐ All references to a variable point to the same memory location.

Reference Chaining

- ☐ We can create a reference to another reference, but all still refer to the original variable.

Example:

```
int a = 10;

int& b = a;

int& c = b;  // [c is still referencing 'a']
```

Memory Aliasing

☐ Since all references point to the same variable, changing one affects all.

☐ int& rnum = num; [rnum is a **reference** to num]
☐ rnum and num are **aliases** (they refer to the **same memory location**).
☐ num, rnum, and r1num **all refer to the same variable**.
☐ Changes the value at the shared memory location to **100**.

Q12.

Find the output of the following code snippet;

```
void update(int&  rnum, int  vnum, int  *pnum)
{
rnum = rnum+500;
vnum = vnum+500;
*pnum = *pnum+500;
}
int main()
{
int num1=11, num2=22,num3=33;
update(num1,num2,&num3);
cout<<num1<<" "<<num2<<" "<<num3<<endl;
return 0;
```

}

Pass by Reference (int& rnum)

- ☐ **Syntax:** int& rnum
- ☐ Any changes made to **rnum directly affect num1** because rnum is just another name for num1.
- ☐ use this when we want the **function to modify the original variable**.
- ☐ rnum = rnum + 500; [changes num1 from 11 to 511]
- ☐ Nickname - Changes affect the person directly.

Pass by Value (int vnum)

- ☐ **Syntax:** int vnum
- ☐ A copy of num2 is made. **Any changes to vnum are local and do not affect num2.**
- ☐ when we want to protect the original value from modification.
- ☐ vnum = vnum + 500; [only changes local copy, num2 remains 22]
- ☐ Photocopy - Writing on it doesn't affect the original.

Pass by Pointer (int* pnum)

- ☐ **Syntax:** int* pnum
- ☐ A pointer to num3 is passed, so **modifying *pnum directly changes num3.**
- ☐ useful when working with multiple values (arrays, dynamic data), want to modify original data
- ☐ *pnum = *pnum + 500; [changes num3 from 33 to 533]

☐ GPS location - You can go and change the actual thing there.

Q13.

Consider the following C++ code snippet;

```
int& getMax(int &a, int &b) {
return (a > b) ? a : b;
}
int main() {
int x=?, y =?;
int& maxVal = getMax(x, y);
cout<<maxVal<<endl;
maxVal = 30;
cout <<"x = "<< x<< ", y= " <<y;
return 0;
}
```

Find the output for given x & y

❑ 10 10 ❑ 20 20 ❑ 10 20 ❑ 20 10 ❑ 60 40 ❑ 40 60

- ☐ **int& maxVal = getMax(x, y);**
- ☐ **maxVal = 30;**
- ☐ maxVal is not just a copy- **it refers to the actual x or y.**
- ☐ maxVal = 30; modifies whichever variable (x or y) was larger.

Case 1: x = 10, y = 10

- ☐ both are equal - returns y by default.
- ☐ Prints: 10
- ☐ Sets y = 30
- ☐ 10
- ☐ x = 10, y = 30

Case 3: x = 10, y = 20

- ☐ y is greater - returns reference to y (y = 30)
- ☐ Prints: 20
- ☐ y = 30

Case 4: x = 20, y = 10

- ☐ x is greater - returns reference to x
- ☐ Prints: 20
- ☐ x = 30
- ☐ 30,10

Q14.

Write SFML-C++ statement **to compute the angle between the player location (x1,y1) to the BLOATER position (x2,y2)**. Additionally, set the rotation of the BLOATER zombie sprite (i.e. m Sprite) to that angle.

- ☐ Computes angle between the bloater and player in radians - **atan2(deltaY, deltaX)**
- ☐ Converts the angle from radians to degrees (for setRotation)- **angle * 180 / π**
- ☐ Rotates the Bloater sprite to face the player - .**setRotation(angleDeg)**

```
#include <SFML/Graphics.hpp>

#include <cmath>

using namespace sf;
```

float x1 = 300; [Player's x position]

float y1 = 200; [Player's y position]

float x2 = 400; [Bloater's x position]

float y2 = 100; [Bloater's y position]

```
while (window.isOpen())
    {
```

```
float deltaX = x1- x2;

float deltaY = y1 - y2;


float Rad = atan2(deltaY, deltaX);


float Deg = Rad * 180 / 3.14159;


bloaterSprite.setRotation(Deg);


window.clear(Color::Black);
window.draw(playerSprite);
window.draw(bloaterSprite);
window.display();
}
```

Q15.

Assume that a zombie sprite, m Sprite, is to the left of the player's position (i.e. Vector2f playerLocation). Write SFML-C++ statement **to update the zombie position variable (m Position) w.r.t. the player.**

SFML C++ Statement to Move Zombie Toward Player

- ***m_Position*** is the zombie's current position
- ***playerLocation*** is the player's current position
- ***m_Speed*** is a float representing how fast the zombie moves [units per second]
- ***dt*** is the delta time used for smooth movement

[compute direction vector from zombie to player]

```
Vector2f  direction = playerLocation - m_Position;
```

[normalize the direction vector]

```
float length = sqrt(direction.x * direction.x + direction.y * direction.y);
if (length != 0)
{
   direction /= length;
}
```

[move zombie towards player]

```
m_Position += direction * m_Speed * dt.asSeconds();
```

[update the zombie's sprite position]

```
m_Sprite.setPosition(m_Position);
```

Q16.

State the code segment to **keep the player (m Position.x & m Position.y) is NOT beyond any of the edges of the current arena** (m Arena) with the surrounding wall tile size, m_TileSize=50. **[A player moving inside an arena, and don't want the player to go outside the visible area (where wall tiles are placed on the edges).]**

If your arena size

- ☐ width = 500, height = 500
- ☐ Tile size = 50

playable area is:

- ☐ Left edge = 50 (wall)
- ☐ Right edge = 500 - 50 = 450
- ☐ Top edge = 50
- ☐ Bottom edge = 500 - 50 = 450

[beyond right edge]

```
if (m_Position.x > m_Arena.width - m_TileSize)
    m_Position.x = m_Arena.width - m_TileSize;
```

[beyond left edge]

```
if (m_Position.x < m_Arena.left + m_TileSize)
    m_Position.x = m_Arena.left + m_TileSize;
```

[beyond bottom edge]

```
if (m_Position.y > m_Arena.height - m_TileSize)
    m_Position.y = m_Arena.height - m_TileSize;
```

[beyond top edge]

```
if (m_Position.y < m_Arena.top + m_TileSize)
    m_Position.y = m_Arena.top + m_TileSize;
```

```
m_Sprite.setPosition(m_Position);
```

Q17.

Write the code segment to generate a random number between 80 and 100.

```
#include <cstdlib>
#include <ctime>

srand(time(0));
int random = rand() % 21 + 80;
```

Q18.

Write the code snippet to create a view from a rectangle as well as a view from its center and size.

```
#include <SFML/Graphics.hpp>

int main()

{

    VideoMode vm(1920,1080);

    RenderWindow window(vm, "View");


    FloatRect  viewRect(0, 0, 960,540);

    View  view(viewRect);


   window.setView(view);

}
```

- ☐ View view;
- ☐ view.setCenter(Vector2f(960,540));
- ☐ view.setSize(Vector2f(1920, 1080));
- ☐ window.setView(view);

Q19.

The view in SFML is like a 2D camera. It controls which part of the 2D scene is visible, and how it is viewed in the render target. The new view will affect everything that is drawn, until another view is set. Write the SFML-C++ statement(s) to set a view to be displayed in the window and draw everything related to it.

```
#include <SFML/Graphics.hpp>

int main()

{

   VideoMode vm(1920, 1080);

    RenderWindow window(vm, "SFML");


    RectangleShape rect(Vector2f(100.f, 100.f));

    rect.setFillColor(Color::Green);

   rect.setPosition(200.f, 200.f);


    View view;

    view.setCenter(Vector2f(960, 540));

    view.setSize(Vector2f(1920, 1080));


    while (window.isOpen()) {
```

```
        Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.setView(view);
        window.clear();
        window.draw(rect);
        window.display();
    }
    return 0;
}
```

Q20.

Design a player class with optimal number of private and public members to draw the player sprite at the center of the defined view of the window.

Bullet Concept

In the Zombie Arena game, the **Bullet class represents the projectiles fired by the player to shoot zombies.**

- ☐ ***bullet.h***

- ☐ Creating bullets with position, shape, speed, and direction.
- ☐ Moving bullets frame-by-frame.
- ☐ Checking whether a bullet is in flight.
- ☐ Stopping bullets when they go out of range or hit a zombie.
- ☐ Rendering the bullet to the screen.

```
#include <SFML/Graphics.hpp>

using namespace sf;

class Bullet

{

private:

    // Where is the bullet?

    Vector2f m_Position;


    // What each bullet looks like

    RectangleShape m_BulletShape;


    // Is this bullet currently whizzing through the air
```

```
bool m_InFlight = false;

// How fast does a bullet travel?
float m_BulletSpeed = 1000;

// What fraction of 1 pixel does the bullet travel,
// Horizontally and vertically each frame?
// These values will be derived from m_BulletSpeed
float m_BulletDistanceX;
float m_BulletDistanceY;

// Where is this bullet headed to?
float m_XTarget;
float m_YTarget;

// Some boundaries so the bullet doesn't fly forever
float m_MaxX;
float m_MinX;
float m_MaxY;
float m_MinY;
```

```
// Public function prototypes go here
public:
	// The constructor
	Bullet();

	// Stop the bullet
	void stop();

	// Returns the value of m_InFlight
	bool isInFlight();

	// Launch a new bullet
	void shoot(float startX, float startY,
		float xTarget, float yTarget);

	// Tell the calling code where the bullet is in the world
	FloatRect getPosition();

	// Return the actual shape (for drawing)
	RectangleShape getShape();
```

```
        // Update the bullet each frame
        void update(float elapsedTime);

};
```

- ***bullet.cpp***

```
        #include "bullet.h"

        // The constructor
        Bullet::Bullet()
        {
                m_BulletShape.setSize(sf::Vector2f(2, 2));
        }

        void Bullet::shoot(float startX, float startY,
                float targetX, float targetY)
        {
                // Keep track of the bullet
                m_InFlight = true;
                m_Position.x = startX;
```

```
m_Position.y = startY;

// Calculate the gradient of the flight path
float gradient = (startX - targetX) / (startY - targetY);

// Any gradient less than zero needs to be negative
if (gradient < 0)
{
        gradient *= -1;
}

// Calculate the ratio between x and t
float ratioXY = m_BulletSpeed / (1 + gradient);

// Set the "speed" horizontally and vertically
m_BulletDistanceY = ratioXY;
m_BulletDistanceX = ratioXY * gradient;

// Point the bullet in the right direction
if (targetX < startX)
{
```

```
        m_BulletDistanceX *= -1;
}

if (targetY < startY)
{
        m_BulletDistanceY *= -1;
}

// Finally, assign the results to the
// member variables
m_XTarget = targetX;
m_YTarget = targetY;

// Set a max range of 1000 pixels
float range = 1000;
m_MinX = startX - range;
m_MaxX = startX + range;
m_MinY = startY - range;
m_MaxY = startY + range;

// Position the bullet ready to be drawn
```

```
	m_BulletShape.setPosition(m_Position);
}

void Bullet::stop()
{
	m_InFlight = false;
}

bool Bullet::isInFlight()
{
	return m_InFlight;
}

FloatRect Bullet::getPosition()
{
	return m_BulletShape.getGlobalBounds();
}

RectangleShape Bullet::getShape()
{
	return m_BulletShape;
```

```
}



void Bullet::update(float elapsedTime)
{
	// Update the bullet position variables
	m_Position.x += m_BulletDistanceX * elapsedTime;
	m_Position.y += m_BulletDistanceY * elapsedTime;

	// Move the bullet
	m_BulletShape.setPosition(m_Position);

	// Has the bullet gone out of range?
	if (m_Position.x < m_MinX || m_Position.x > m_MaxX ||
		m_Position.y < m_MinY || m_Position.y > m_MaxY)
	{
		m_InFlight = false;
	}
}
```

Q1.

Write the SFML-C++ statement(s) to set the position (i.e. m Position) of a bullet with the parameters **startX** and **startY.** Also calculate the gradient of travel for a bullet to the target **targetX** and **targetY**.

- ☐ When the player fires a bullet, it travels **from the player's position** (`startX`, `startY`) to the **target position** (`targetX`, `targetY`) where the mouse was clicked.

(startX, startY) (targetX, targetY)

Player ---------- Mouse Click

[*] [X]

Bullet start Bullet target

// Set the initial position of the bullet

m_Position.x = startX;

m_Position.y = startY;

m_BulletShape.setPosition(m_Position);

// Calculate the gradient of the flight path

float gradient = (startX - targetX) / (startY - targetY);

if (gradient < 0)

```
{
   gradient *= -1;
}
```

Q2.

Assume that the gradient of the flight path is float gradient; as calculated in the previous question. Now **compute the speed horizontally and vertically for the bullet in terms of gradient given the bullet speed as m BulletSpeed.**

m BulletSpeedY =

m BulletSpeedX =

The private member variables ***m BulletSpeedY & m BulletSpeedX*** can also be stated as ***m BulletDistanceY & m BulletDistanceX*** respectively.

Ans

```
float gradient;  [already computed from (startX - targetX) / (startY - targetY)]
float m_BulletSpeed = 1000;


m_BulletDistanceY = m_BulletSpeed / (1 + gradient);
m_BulletDistanceX = m_BulletDistanceY * gradient;
```

Q3.

Write the code snippet to set a ***maximum horizontal and vertical location that the bullet can reach from the position startX and startY***, assuming a maximum range of 1200 pixels in any direction a bullet can be fired.

[Inside the shoot() function]

```
float range = 1200.0f;
```

*[**Define a bounding box within which the bullet can travel** : (m_MinX, m_MaxX, m_MinY, m_MaxY)]*

```
m_MinX = startX - range;

m_MaxX = startX + range;

m_MinY = startY - range;

m_MaxY = startY + range;
```

[update() function, this boundary helps stop the bullet if it goes beyond range]

```
if (m_Position.x < m_MinX || m_Position.x > m_MaxX ||
   m_Position.y < m_MinY || m_Position.y > m_MaxY)
{
   m_InFlight = false;
}
```

Q4.

Write the code snippet to test whether the bullet has moved beyond its maximum range. If so, set m **InFlight=false**.

```
if (m_Position.x < m_MinX || m_Position.x > m_MaxX ||
   m_Position.y < m_MinY || m_Position.y > m_MaxY)
{
   m_InFlight = false;
}
```

Q5.

Design the code snippet to handle the left mouse button being clicked to ***fire a bullet.*** Also identify the area in our main program to place this mouse handle part.

```
if (Mouse::isButtonPressed(Mouse::Left))
{
   if (gameTimeTotal.asMilliseconds() - lastPressed.asMilliseconds() > 1000 / fireRate && bulletsInClip > 0)
   {
      bullets[currentBullet].shoot(player.getCenter().x, player.getCenter().y,
         mouseWorldPosition.x, mouseWorldPosition.y );
```

```
        currentBullet++;
        if (currentBullet > maxBullets - 1)
            currentBullet = 0;

        lastPressed = gameTimeTotal;
        bulletsInClip--;
    }
}
```

Inheritance

Q1.

Let say a class B is publicly derived from class A with Class A has 4 public members and class B has 1 public member. Write the C++ syntax for such derivation along with a number of public functions for class B to use.

Logic:

```
class A
{
public:
    void func1();
```

```
    void func2();
    void func3();
    void func4();
};

class B : public A
{
public:
    void func5();
};

int main()
{
    B obj;
    obj.func1();
}
```

Q2.

Let say a class B is privately derived from class A with Class A has 4 public members and class B has 1 private member. Write the C++ syntax for such derivation along with a number of private members for class B to use.

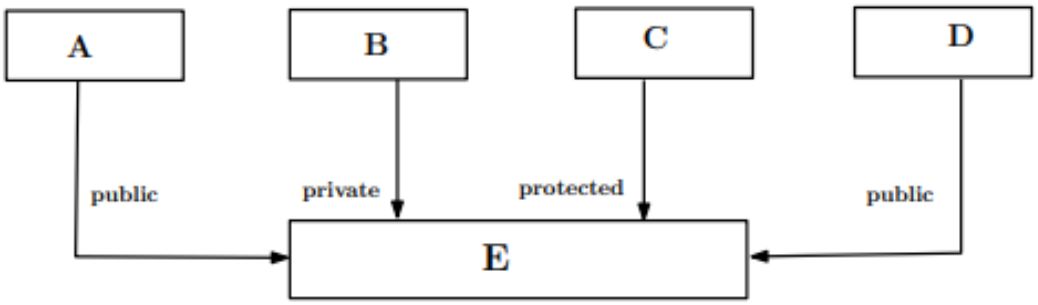
```
#include <iostream>

using namespace std;


class A

{

public:

    void func1() { cout << "func1 from A"; }

    void func2() { cout << "func2 from A"; }

    void func3() { cout << "func3 from A"; }

    void func4() { cout << "func4 from A"; }

};


class B : private A

{

private:

    int Data = 10;
```

```
public:
    void access() {
        func1();
        func2();
        func3();
        func4();
    }

    void sData() {
        cout << "Private data: " << Data << endl;
    }
};

int main()
{
    B obj;

    //obj.func1();
    obj.access();
```

```
    obj.sData();
    return 0;
}
```

Q3.

Write the C++ syntax for class E inherited from classes A, B, C and D. The figure shows the type of derivation.




```
#include <iostream>
using namespace std;

class A
{
public:
    void showA() {
        cout << "A Public Method";
    }
```

```
};

class B
{
public:
    void showB() {
        cout << "B Public Method";
    }
};

class C
{
public:
    void showC() {
        cout << "C Public Method";
    }
};

class D
{
public:
```

```
    void showD() {
        cout << "D Public Method";
    }
};

class E : public A, private B, protected C, public D
{
public:
    void showAllAccessible() {
        showA(); // Accessible: public in A → public in E
        showB(); // Accessible: public in B → private in E
        showC(); // Accessible: public in C → protected in E
        showD(); // Accessible: public in D → public in E
    }
};

int main() {
    E obj;
    obj.showA();  // yes
    obj.showD();  // yes
```

```
    // obj.showB(); // no
    // obj.showC(); // no

    obj.showAllAccessible();
    return 0;
}
```

Q4.

Find the output of the following code snippet;

```
class B1 {
 public:B1(){cout <<"B1"<<endl;}
};
class B2 {
 public:B2(){cout<<"B 2"<<endl;}
};
class : public B1, public B2 {
   public:D(){cout << "D" <<endl;  }
};
int main(){
   Derived d;return 0;}
```

☐ **multiple inheritance - Order of constructor calls**.

☐ When creating an object of Derived, the **base class constructors** are called **first**, and **in the order of inheritance**.

☐ B1

☐ B2

☐ D