**Estructura de Juego:**

**Análisis Detallado de las Funciones**

**1. Main(string[] args)**

* **Lógica**: Esta es la función principal del programa. Simplemente llama a IniciarJuego() para comenzar el juego. No contiene lógica adicional, ya que su único propósito es iniciar el flujo del programa.

**2. ReStartStatusGame()**

* **Lógica**: Esta función reinicia todas las variables globales a sus valores iniciales. Esto incluye:
  + Restablecer la salud y velocidad de los jugadores.
  + Reiniciar los cooldowns de habilidades.
  + Limpiar los diccionarios de habilidades y cooldowns.
  + Reiniciar los niveles y la experiencia de los jugadores.
* **Propósito**: Asegura que el juego comience en un estado limpio y consistente cada vez que se reinicia.

**3. IniciarJuego()**

* **Lógica**:
  1. Llama a ReStartStatusGame() para reiniciar las estadísticas.
  2. Llama a SeleccionPlayer() para que los jugadores elijan sus personajes.
  3. Crea el tablero con CrearTablero().
  4. Coloca cofres, cofres raros, trampas y la meta en el tablero.
  5. Coloca a los jugadores en posiciones aleatorias con ColocarPlayer().
  6. Entra en un bucle principal donde alterna los turnos de los jugadores usando TurnoJugador().
  7. Después de cada turno, reduce los cooldowns de las habilidades si es necesario.
* **Propósito**: Controla el flujo principal del juego, desde la inicialización hasta la ejecución de los turnos.

**4. MostrarInstrucciones()**

* **Lógica**: Muestra un mensaje en la consola con las instrucciones del juego. Esto incluye:
  + Cómo moverse.
  + Cómo usar habilidades.
  + La simbología del tablero (muros, cofres, trampas, etc.).
* **Propósito**: Proporciona al jugador la información necesaria para entender y jugar el juego.

**5. SeleccionPlayer()**

* **Lógica**:
  1. Muestra un mensaje de introducción al juego.
  2. Pide al Jugador 1 que seleccione un personaje.
  3. Valida la entrada del Jugador 1. Si no es válida, asigna el personaje "Ninguno" como castigo.
  4. Repite el proceso para el Jugador 2, asegurándose de que no seleccione el mismo personaje que el Jugador 1.
  5. Asigna las habilidades y cooldowns correspondientes usando AsignarPlayer().
* **Propósito**: Permite a los jugadores elegir sus personajes y asigna las habilidades correspondientes.

**6. AsignarPlayer(int opcion, Dictionary<string, Action> skills, Dictionary<string, int> cooldowns)**

* **Lógica**:
  1. Dependiendo de la opción seleccionada, asigna una habilidad y un cooldown al jugador.
  2. Las habilidades se almacenan en el diccionario skills como acciones (Action).
  3. Los cooldowns se almacenan en el diccionario cooldowns.
  4. Si la opción no es válida, asigna el personaje "Ninguno" con una habilidad que penaliza al jugador.
* **Propósito**: Asigna las habilidades y cooldowns correspondientes al personaje seleccionado.

**7. Teleport(ref int positionX, ref int positionY, bool isPlayer1)**

* **Lógica**:
  1. Genera coordenadas aleatorias dentro del tablero.
  2. Verifica que la nueva posición esté vacía (' ').
  3. Actualiza las coordenadas del jugador a la nueva posición.
  4. Muestra un mensaje indicando que el jugador se ha teletransportado.
* **Propósito**: Teletransporta al jugador a una ubicación aleatoria en el tablero.

**8. WarriorDamage(bool isPlayer1)**

* **Lógica**:
  1. Verifica si el otro jugador está cerca (a una distancia de 1 celda).
  2. Si está cerca, calcula un daño aleatorio entre 50 y 100 (para el Jugador 1) o entre 40 y 61 (para el Jugador 2).
  3. Reduce la salud del otro jugador y verifica si ha llegado a 0.
  4. Si la salud llega a 0, finaliza el juego con EndGame().
* **Propósito**: Inflige daño al otro jugador si está cerca.

**9. WhitcherDamage(bool isPlayer1)**

* **Lógica**:
  1. Calcula un daño aleatorio entre 10 y 21.
  2. Reduce la salud del otro jugador y verifica si ha llegado a 0.
  3. Si la salud llega a 0, finaliza el juego con EndGame().
* **Propósito**: Inflige daño al otro jugador sin necesidad de estar cerca.

**10. NoneSkill(bool isPlayer1)**

* **Lógica**:
  1. Muestra un mensaje de castigo.
  2. Reduce la salud y la velocidad del jugador como penalización.
* **Propósito**: Penaliza al jugador por no seleccionar un personaje válido.

**11. GodSkill(bool isPlayer1)**

* **Lógica**:
  1. Muestra un mensaje indicando que el jugador ha usado la habilidad de "Dios".
  2. Finaliza el juego inmediatamente con EndGame(), declarando al jugador como ganador.
* **Propósito**: Finaliza el juego de manera instantánea.

**12. RevelarMeta()**

* **Lógica**:
  1. Busca la posición de la meta ('M') en el tablero.
  2. Muestra las coordenadas de la meta en la consola.
* **Propósito**: Revela la ubicación de la meta al jugador.

**13. TeleportToDugeon()**

* **Lógica**:
  1. Muestra un mensaje indicando que los jugadores han sido transportados a una nueva mazmorra.
  2. Llama a CrearTablero(), ColocarCofres(), ColocarRareCofres(), ColocarTrampas(), ColocarMeta() y ColocarPlayer() para generar un nuevo tablero.
* **Propósito**: Transporta a los jugadores a una nueva mazmorra.

**14. RestartGame()**

* **Lógica**:
  1. Muestra un mensaje indicando que el juego se reiniciará.
  2. Llama a IniciarJuego() para reiniciar el juego.
* **Propósito**: Reinicia el juego completamente.

**15. ResetPlayerStats()**

* **Lógica**:
  1. Restablece las estadísticas de los jugadores a sus valores iniciales (salud, velocidad, nivel, experiencia).
* **Propósito**: Reinicia las estadísticas de los jugadores.

**16. CrearTablero()**

* **Lógica**:
  1. Inicializa todas las celdas del tablero como muros (#).
  2. Llama a GenerarLaberinto() para crear pasajes y espacios vacíos.
* **Propósito**: Crea el tablero inicial con muros y pasajes.

**17. GenerarLaberinto(int x, int y)**

* **Lógica**:
  1. Utiliza el algoritmo de backtracking para generar un laberinto.
  2. Desde la posición inicial (x, y), elige direcciones aleatorias y abre pasajes.
  3. Repite el proceso recursivamente hasta que no queden más celdas por visitar.
* **Propósito**: Genera un laberinto aleatorio en el tablero.

**18. ColocarCofres(int cantidadCofres)**

* **Lógica**:
  1. Recorre el tablero y recopila todas las celdas vacías.
  2. Selecciona aleatoriamente cantidadCofres celdas y coloca cofres ('C') en ellas.
* **Propósito**: Coloca cofres normales en el tablero.

**19. ProcesamientoCofre(bool isPlayer1)**

* **Lógica**:
  1. Genera un número aleatorio para determinar el efecto del cofre.
  2. Aplica el efecto correspondiente (aumentar/disminuir velocidad, salud, o causar daño).
  3. Llama a GanarExperiencia() para otorgar experiencia al jugador.
* **Propósito**: Procesa el efecto de abrir un cofre normal.

**20. ColocarRareCofres(int CantidadCofres)**

* **Lógica**:
  1. Similar a ColocarCofres(), pero coloca cofres raros ('R').
* **Propósito**: Coloca cofres raros en el tablero.

**21. ProcesamientoRareCofre(bool isplayer1)**

* **Lógica**:
  1. Similar a ProcesamientoCofre(), pero con efectos más extremos.
* **Propósito**: Procesa el efecto de abrir un cofre raro.

**22. ColocarTrampas(int cantidadTrap)**

* **Lógica**:
  1. Recorre el tablero y recopila todas las celdas vacías.
  2. Selecciona aleatoriamente cantidadTrap celdas y coloca trampas ('T') en ellas.
* **Propósito**: Coloca trampas en el tablero.

**23. ColocarMeta(int meta)**

* **Lógica**:
  1. Recorre el tablero y recopila todas las celdas vacías.
  2. Selecciona aleatoriamente una celda y coloca la meta ('M') en ella.
* **Propósito**: Coloca la meta en el tablero.

**24. ColocarPlayer()**

* **Lógica**:
  1. Genera coordenadas aleatorias para el Jugador 1 y el Jugador 2.
  2. Asegura que los jugadores no estén en la misma celda.
* **Propósito**: Coloca a los jugadores en el tablero.

**25. MostrarDungeon()**

* **Lógica**:
  1. Recorre el tablero y muestra cada celda en la consola.
  2. Resalta las posiciones de los jugadores y los elementos del tablero con colores.
* **Propósito**: Muestra el tablero en la consola.

**26. MostrarElemento(char elemento)**

* **Lógica**:
  1. Dependiendo del carácter, muestra el elemento con un color específico.
* **Propósito**: Muestra un elemento del tablero con colores.

**27. MostrarEstadisticasJugadores()**

* **Lógica**:
  1. Muestra las estadísticas de los jugadores, como nivel, experiencia, velocidad y salud.
* **Propósito**: Proporciona información sobre el estado de los jugadores.

**28. TurnoJugador(bool Player1)**

* **Lógica**:
  1. Calcula el número máximo de movimientos que el jugador puede realizar en su turno.
  2. Permite al jugador moverse o usar una habilidad.
  3. Verifica si el jugador ha llegado a la meta o ha caído en una trampa.
* **Propósito**: Maneja el turno de un jugador.

**29. UsarHabilidad(bool isPlayer1)**

* **Lógica**:
  1. Dependiendo del personaje seleccionado, ejecuta la habilidad correspondiente.
  2. Aplica el efecto de la habilidad y establece el cooldown.
* **Propósito**: Ejecuta la habilidad del jugador.

**30. ValidMove(int newX, int newY)**

* **Lógica**:
  1. Verifica si la nueva posición está dentro del tablero.
  2. Verifica que la nueva posición no sea un muro (#).
  3. Verifica que la nueva posición no esté ocupada por otro jugador.
* **Propósito**: Verifica si un movimiento es válido.

**31. Move(int posicionX, int posicionY, bool Player1)**

* **Lógica**:
  1. Calcula la nueva posición del jugador.
  2. Verifica si el movimiento es válido con ValidMove().
  3. Si el jugador cae en una trampa, abre un cofre o llega a la meta, procesa el evento correspondiente.
* **Propósito**: Mueve al jugador en la dirección especificada.

**32. EndGame(string Message)**

* **Lógica**:
  1. Muestra un mensaje de victoria o derrota.
  2. Permite al jugador reiniciar el juego o salir.
* **Propósito**: Finaliza el juego y maneja las opciones posteriores.

**33. GanarExperiencia(bool esJugador1, int cantidad)**

* **Lógica**:
  1. Aumenta la experiencia del jugador.
  2. Si la experiencia alcanza el umbral necesario, sube de nivel y mejora sus estadísticas.
* **Propósito**: Gestiona la experiencia y el nivel de los jugadores.

**34. ReproducirSonido(string sonido)**

* **Lógica**:
  1. Reproduce un sonido en la consola usando Console.Beep().
* **Propósito**: Proporciona retroalimentación auditiva al jugador.

**35. CheckVictoryCondition()**

* **Lógica**:
  1. Verifica si la salud de algún jugador ha llegado a 0.
  2. Si es así, finaliza el juego con EndGame().
* **Propósito**: Verifica si se ha cumplido alguna condición de victoria.

Proceso de Creación:

**1. La Idea y la Planificación**

Cuando me propusieron crear un juego, lo primero que pensé fue: "¿Cómo empiezo?" Sabía que quería hacer un laberinto con mecánicas de exploración y turnos, pero no tenía mucha experiencia en programación. Así que me senté a investigar. Decidí que lo más complicado sería generar un laberinto aleatorio donde todos los caminos estuvieran conectados. Después de buscar un poco, encontré el algoritmo DFS (Depth-First Search), que parecía perfecto para lo que necesitaba.

**Mi pensamiento:** "Si puedo generar un laberinto conectado, el resto será más fácil. Voy a empezar por ahí."

**2. Generando el Laberinto**

Implementé el DFS para crear el laberinto. La idea era que el algoritmo recorriera el tablero, marcara los caminos válidos y convirtiera el resto en muros. Al principio, todo parecía funcionar bien. El laberinto se generaba, y los caminos estaban conectados. Pero luego me di cuenta de un problema: a veces no quedaba suficiente espacio para colocar cofres, trampas y otros elementos. ¡Eso no podía pasar!

**Mi solución:** Cambié el método. En lugar de usar DFS, implementé un sistema que crea caminos a partir de una matriz llena de muros. Así me aseguré de que siempre hubiera espacio para todo. Fue un cambio de última hora, pero funcionó.

**Mi reflexión:** "A veces las cosas no salen como las planeas, pero siempre hay una solución. Solo hay que buscarla."

**3. Las Mecánicas del Juego**

Una vez que el laberinto estaba listo, me puse a trabajar en las mecánicas del juego. Creé una función llamada TurnoJugador que controla cuánto se puede mover cada jugador según su velocidad. Luego implementé la función Move, que maneja lo que pasa cuando un jugador se mueve a una casilla: si hay una trampa, un cofre, la meta, o simplemente un espacio vacío. También añadí ValidMove para asegurarme de que los jugadores no se salieran del laberinto o se movieran a lugares inválidos.

**Mi pensamiento:** "Quiero que el movimiento sea justo y que los jugadores sientan que cada decisión cuenta."

**4. Las Clases y Habilidades**

Aquí fue donde me metí en algo más complicado. Quería que los jugadores pudieran elegir entre diferentes clases, cada una con habilidades únicas. Creé diccionarios para almacenar las habilidades y los tiempos de enfriamiento de cada clase (Paladín, Mago, Guerrero, Hechicero). Luego implementé un sistema para que los jugadores seleccionaran su clase y usaran habilidades durante el juego.

Pero me encontré con un error: al usar una habilidad, el código verificaba todas las clases en lugar de solo la clase seleccionada. ¡Eso me volvió loco! Pasé horas intentando encontrar el problema, hasta que me di cuenta de que tenía que poner la verificación dentro de una condicional para cada jugador.

**Mi reflexión:** "Los errores de lógica son los más difíciles de encontrar, pero también son los que más te enseñan. Cada error que resuelves te hace mejor programador."

**5. Mejoras y Detalles Finales**

Cuando el juego ya funcionaba, me puse a mejorarlo. Añadí funciones como verificarGanador y finalizarJuego, que son esenciales para que el juego sea completo. También manejé casos en los que los jugadores ingresaban letras en lugar de números, lo cual es importante para evitar errores. Además, incluí clases ocultas y una clase difícil de desbloquear, lo que añade un toque de misterio y rejugabilidad.

**Mi pensamiento:** "Quiero que el juego sea divertido y que los jugadores sientan que hay cosas por descubrir."

**6. El Error de Última Hora**

Casi al final, me di cuenta de que el método DFS no siempre dejaba suficiente espacio para los elementos del juego. Esto fue un problema serio, pero no me rendí. En lugar de eso, implementé un nuevo método que crea caminos a partir de una matriz llena de muros, asegurándome de que siempre haya espacio para todo. ¡Eso fue un gran logro!

**Mi reflexión:** "A veces los errores más grandes aparecen al final, pero eso no significa que no puedas solucionarlos. La clave es mantener la calma y buscar una solución."

**7. Lo que Aprendí**

Este proyecto me enseñó muchas cosas:

* **Planificar es esencial:** Investigar y tener un plan desde el principio te ahorra muchos problemas.
* **Los errores son oportunidades:** Cada error que resolví me hizo mejor programador.
* **La persistencia es clave:** Aunque hubo momentos difíciles, no me rendí y logré terminar el juego.
* **Mejorar es un proceso:** Añadir funciones adicionales y manejar casos especiales hace que el juego sea más completo y profesional.

**Consejos para Futuros Proyectos**

* **Divide el código en clases:** A medida que tus proyectos crezcan, intenta separar el código en clases (por ejemplo, una clase para el laberinto, otra para los jugadores, etc.). Esto hará que el código sea más organizado y fácil de mantener.
* **Prueba todo:** Dedica tiempo a probar cada parte del juego, especialmente las mecánicas principales. Esto te ayudará a encontrar errores antes de que se conviertan en problemas grandes.
* **Documenta tu código:** Añade comentarios y documentación para que tú (y otros) puedan entender fácilmente cómo funciona el código.
* **No temas refactorizar:** Si encuentras una manera mejor de hacer algo, no dudes en cambiar el código. Refactorizar es parte del proceso de aprendizaje.

**Conclusión**

Crear este juego fue un desafío enorme, pero también una experiencia increíble. Aprendí mucho sobre programación, resolución de problemas y perseverancia. Cada error que superé y cada función que añadí me hicieron sentir más seguro de mis habilidades. ¡Estoy orgulloso del resultado final! Y lo más importante: sé que el próximo proyecto será aún mejor.