

República Bolivariana de Venezuela
Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Asignatura: Algoritmos y Programación

INFORME

DEL PROYECTO #1

(explicación de la implementación, metodología, análisis y enfoque)

Profesor

Ricardo Osuna

Estudiantes

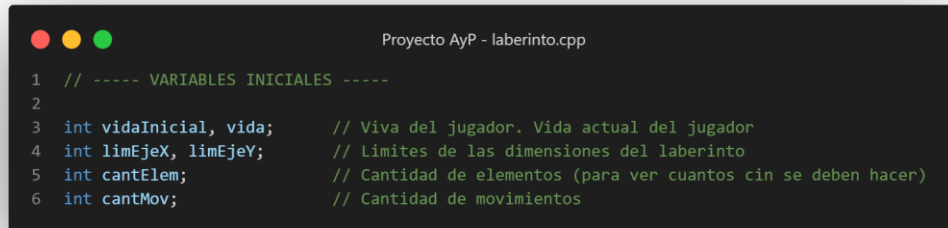
José Maceiras
C.I. 31.228.863

Leonardo Espinola
C.I. 32.105.059

Caracas, 8 de enero de 2025

El siguiente informe tiene como propósito explicar cómo hemos desarrollado el programa requerido y como hemos resuelto cada problema presentado

Cuando se empezó a desarrollar el proyecto, tuvimos que crear algunas variables globales para poder guardar los datos más básicos del laberinto que el usuario puede introducir. Estas variables fueron las siguientes:



```
1 // ----- VARIABLES INICIALES -----
2
3 int vidaInicial, vida;           // Vida del jugador. Vida actual del jugador
4 int limEjeX, limEjeY;           // Limites de las dimensiones del laberinto
5 int cantElem;                   // Cantidad de elementos (para ver cuantos cin se deben hacer)
6 int cantMov;                    // Cantidad de movimientos
```

Estas variables son útiles a lo largo de todo el código y son fundamentales en la siguiente parte que se desarrolló, que fue la entrada del usuario. Toda la entrada del usuario y el algoritmo que gestiona las coordenadas de los elementos en las celdas se encuentran en la función **inicializar()**. Se decidió intentar hacer el código lo más modular posible usando funciones desde un inicio para poder gestionar todo de una manera más eficiente.

A la hora de elaborar este proyecto nos presentamos con algunas limitantes, como que no podemos usar arreglos, que hubieran sido más útiles y eficientes a la hora de guardar datos en las variables, por lo que implementamos un sistema para guardar las variables de la única manera posible: declarando un montón de variables separadas.

Creamos el sistema para guardar los datos de las coordenadas X y Y de los portales, trampas, muros, tesoros (son aproximadamente 20 declaraciones de variables cada uno), entrada y salida (4 declaraciones variables), posición actual del jugador (2 variables), llevar la cuenta de cuántos elementos hay en el laberinto y las estadísticas del jugador (trampas activadas, tesoros descubiertos, movimientos bloqueados). Todas estas variables son globales.

```

Proyecto AyP - laberinto.cpp

1 // Variables de los portales
2 int portal1_x, portal1_y, portal2_x, portal2_y, portal3_x, portal3_y, portal4_x, portal4_y, portal5_x, po
rtal5_y, portal6_x, portal6_y, portal7_x, portal7_y, portal8_x, portal8_y, portal9_x, portal9_y, portal10
_x, portal10_y;
3
4 // Variables de los portales vinculados
5 int portalVinc1_x, portalVinc1_y, portalVinc2_x, portalVinc2_y, portalVinc3_x, portalVinc3_y, portalVinc4
_x, portalVinc4_y, portalVinc5_x, portalVinc5_y, portalVinc6_x, portalVinc6_y, portalVinc7_x, portalVinc7
_y, portalVinc8_x, portalVinc8_y, portalVinc9_x, portalVinc9_y, portalVinc10_x, portalVinc10_y;
6
7 // Variables de las Trampas
8 int trampa1_x, trampa1_y, trampa2_x, trampa2_y, trampa3_x, trampa3_y, trampa4_x, trampa4_y, trampa5_x, tr
ampa5_y, trampa6_x, trampa6_y, trampa7_x, trampa7_y, trampa8_x, trampa8_y, trampa9_x, trampa9_y, trampa10
_x, trampa10_y;
9
10 // Variables de los muros
11 int muro1_x, muro1_y, muro2_x, muro2_y, muro3_x, muro3_y, muro4_x, muro4_y, muro5_x, muro5_y, muro6_x, mu
ro6_y, muro7_x, muro7_y, muro8_x, muro8_y, muro9_x, muro9_y, muro10_x, muro10_y;
12
13 // Variables de los tesoros
14 int tesoro1_x, tesoro1_y, tesoro2_x, tesoro2_y, tesoro3_x, tesoro3_y, tesoro4_x, tesoro4_y, tesoro5_x, te
soro5_y, tesoro6_x, tesoro6_y, tesoro7_x, tesoro7_y, tesoro8_x, tesoro8_y, tesoro9_x, tesoro9_y, tesoro10
_x, tesoro10_y;
15
16 // Coordenadas de las entradas y salidas del laberinto
17 int entrada_x, entrada_y, salida_x, salida_y;
18
19 // Posicion Actual del jugador
20 int player_pos_x, player_pos_y;
21
22 // Variables para llevar la cuenta de cada cosa en el laberinto.
23 int cantPortales, cantTrampas, cantMuros, cantTesoros, cantEntradas, cantSalidas;
24
25 // Estadísticas del jugador
26 int cantTrampas_activadas = 0, cantTesoros_activados = 0, cantMovBloqueados = 0;

```

Las primeras variables que lee el programa son la vida inicial, los límites del laberinto, y la cantidad de elementos del mapa.

Dependiendo del número de la cantidad de elementos, el programa hará un bucle para leer las siguientes líneas (la introducción de las coordenadas de los elementos).

Para guardar cada uno de los objetos introducimos en la entrada seguimos el siguiente algoritmo:

Se ingresa un dato en un formato tipo “Letra código” “Coordenada X” “Coordenada Y”. La letra código se guarda dentro de la variable “tipo” (char) y las coordenadas dentro de las variables “posX” y “posY”. Se comprueba la variable “tipo” con un switch. Por ejemplo, si se introduce “#” las coordenadas X y Y se asignan a el objeto muro que corresponda y siempre y cuando no haya más de 10 objetos del mismo tipo, también se le sumara 1 a la cantidad de

```

Proyecto AyP - laberinto.cpp

1 case '#':
2     if (cantMuros < 10) {
3         cantMuros++;
4         switch (cantMuros) {
5             case 1: muro1_x = posX; muro1_y = posY; break;
6             case 2: muro2_x = posX; muro2_y = posY; break;
7             case 3: muro3_x = posX; muro3_y = posY; break;
8             case 4: muro4_x = posX; muro4_y = posY; break;
9             case 5: muro5_x = posX; muro5_y = posY; break;
10            case 6: muro6_x = posX; muro6_y = posY; break;
11            case 7: muro7_x = posX; muro7_y = posY; break;
12            case 8: muro8_x = posX; muro8_y = posY; break;
13            case 9: muro9_x = posX; muro9_y = posY; break;
14            case 10: muro10_x = posX; muro10_y = posY; break;
15        }
16        //cout << "Muro en (" << posX << ", " << posY << ")" << endl; // Q
17    }
18    break;

```

muros. Todo esto se controla con un condicional que compruebe la variable que guarda la cantidad de muros que hay. Todos los ingresos de datos siguen esta misma estructura.

Los portales son los únicos que se guardan de forma distinta, ya que se guardan en pares con 4 coordenadas, por lo que se leen 2 coordenadas más, que son pos2X y pos2Y.



```
Proyecto AyP - laberinto.cpp
1 case 'P':
2     cin >> pos2X >> pos2Y;
3     if (cantPortales < 10) {
4         cantPortales++;
5         i++;
6         switch (cantPortales) {
7             case 1:
8                 portal1_x = posX; portal1_y = posY;
9                 portalVinci_x = pos2X; portalVinci_y = pos2Y;
10                break;
```

Luego de tener la entrada ya funcional, comenzamos con las funciones específicas que manejan los requerimientos del laberinto. Decidimos empezar creando 2 funciones, una para agregarle vida al jugador, que llamamos **sumarVida(<vida a añadir>)**; esta función tiene como propósito comprobar la vida actual del jugador y evaluar si se le puede sumar la vida indicada, sin exceder nunca la vida inicial del jugador.



```
Proyecto AyP - laberinto.cpp
1 void sumarVida(int vidaAnadida) {
2     int chequeoVida = vida + vidaAnadida;
3     if (chequeoVida > vidaInicial) {
4         vida = vidaInicial;
5     }
6     else {
7         vida = chequeoVida;
8     }
9 }
```

Y también existe la función **restarVida(<vida a restar>)**; esta función se limita a restar la vida indicada, no hace ninguna comprobación si es igual a 0 o menor (eso se explicará más adelante).

Las siguientes funciones comparten recibir como entrada la posición del jugador y recorren en un bucle que comprueba la cantidad de todos los objetos que le corresponda a cada una de las funciones en el mapa y comprobando si alguno de esos objetos tiene la misma posX y posY que el jugador, en ese caso ejecuta las siguientes acciones dependiendo de cada función.

- **comprobarTrampa(<posX del jugador>, <posY del jugador>)** solo resta 10 puntos de vida utilizando la función `restarVida(10)`.

```
Proyecto AyP - laberinto.cpp
1 void comprobarTrampa(int posX, int posY) {
2     for(int i = 1; i <= cantTrampas; i++) {
3         switch (i) {
4             case 1:
5                 if(posX == trampa1_x && posY == trampa1_y) {
6                     restarVida(10);
7                     cantTrampas_activadas++;
8                 }
9                 break;
10    }
```

- **comprobarTesoro(<posX del jugador>, <posY del jugador>)** ejecuta la función `sumarVida()` y reposiciona el tesoro a la coordenada x=999 y=999 esto para evitar que el tesoro pueda volver a ser activado. Se decidió enviarlos a esa coordenada ya que el laberinto siempre está acotado por 9 en ambos ejes.

```
Proyecto AyP - laberinto.cpp
1 if(posX == tesoro1_x && posY == tesoro1_y) {
2     tesoro1_x = 999;
3     tesoro1_y = 999;
4     sumarVida(20);
5     cantTesoros_activados++;
6 }
```

- **comprobarPortales(<posX del jugador>, <posY del jugador>)** asigna como posición del jugador las coordenadas del portal vinculado.


```
Proyecto AyP - laberinto.cpp
1 if(posX == portal1_x && posY == portal1_y) {
2     player_pos_x = portalVinc1_x;
3     player_pos_y = portalVinc1_y;
4 }
5 else if(posX == portalVinc1_x && posY == portalVinc1_y){
6     player_pos_x = portal1_x;
7     player_pos_y = portal1_y;
8 }
9 break;
```

- **comprobarMuro(<posX del jugador>, <posY del jugador>)** devuelve true si hay un muro.

```
Proyecto AyP - laberinto.cpp
1 if(posX == muro1_x && posY == muro1_y) {
2     return true;
3 }
```

Para terminar, dentro del **main()** se encuentra el ciclo de ejecución el cual se encarga de mover la partida y es donde ingresas la dirección en la que se moverá el jugador.

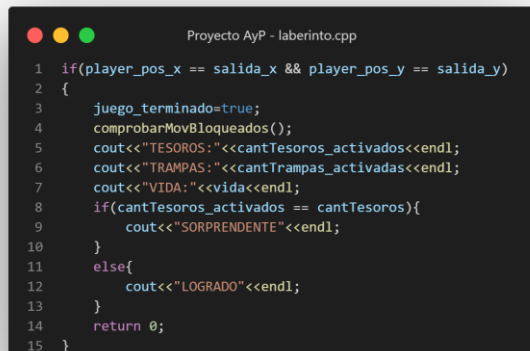
Como ejemplo si ingresamos “d” y nuestra posición x es menor al límite del mapa comprobaremos con una variable que se llama x_deseada que hacia donde nos queremos mover no haya un muro, entonces llamamos a la función de comprobarMuro() y le pasamos como dato x_deseada (o la posición x del jugador+1) en caso de que ésta retorne true no podemos sumarle 1 en la posición x, de lo contrario si avanzamos, y así con cada una de las 4 posibles direcciones.



```
1  else if(direction=='d' && player_pos_x < limEjeX)
2  {
3      x_deseada = player_pos_x + 1;
4      if(!comprobarMuro(x_deseada, player_pos_y)){
5          player_pos_x++;
6      }
7      else{
8          //cout<<"Movimiento bloqueado"<<endl;
9          cantMovBloqueados++;
10     }
11     cantMov--;
12 }
```

También en el main se hacen los llamados a las funciones para comprobar si el jugador está en alguna casilla de algún elemento.

El ciclo de ejecución cuenta con 3 condiciones de salida, si el jugador se queda sin vida, si el jugador se queda sin movimientos o si el jugador llega a la salida se detendrá la ejecución del programa y se mostrará por consola los movimientos que fueron bloqueados, la vida del jugador, la cantidad de trampas activadas, la cantidad de tesoros activados, y un mensaje que depende de tus resultados durante el juego.



```
1  if(player_pos_x == salida_x && player_pos_y == salida_y)
2  {
3      juego_terminado = true;
4      comprobarMovBloqueados();
5      cout<<"TESOROS:"<<cantTesoros_activados<<endl;
6      cout<<"TRAMPAS:"<<cantTrampas_activadas<<endl;
7      cout<<"VIDA:"<<vida<<endl;
8      if(cantTesoros_activados == cantTesoros){
9          cout<<"SORPRELENDE"<<endl;
10     }
11     else{
12         cout<<"LOGRADO"<<endl;
13     }
14     return 0;
15 }
```

Parte del código que muestra la salida cuando el jugador logra salir del laberinto.