Proyecto FINAL: Generación y Resolución de un Laberinto

Objetivos.

En este proyecto de laboratorio se pretende que el alumno implemente un programa en C++ haciendo uso de tipos estructurados. Además, debe ser capaz de aplicar la programación modular.

A tener en cuenta:

1. La entrega se realizará, como muy tarde, el día del examen de INF hasta la medianoche.
2. Se entrega por parejas (con la persona con la que se han realizado las prácticas) o individualmente.
3. El proyecto se expondrá ante el profesor de laboratorio. El profesor citará a los alumnos con suficiente antelación. El profesor puede solicitar a los estudiantes que (por parejas o de forma individual) amplíen o modifiquen en ese momento el código (oral o por escrito).
4. En el examen final de la asignatura aparecerá una pregunta sobre el proyecto final.
5. Es recomendable realizar una tutoría.
6. Se deben **seguir estrictamente** los requisitos y especificaciones que se indican en el enunciado.
7. Se debe adjuntar un documento (.pdf) con una breve explicación de:
   - Las definiciones de tipos utilizados. Justificación de su elección.
   - La implementación de los subprogramas: descripción y justificación de los parámetros de entrada y de salida, tipo de paso de parámetros y tarea que realiza cada función.
   - El diagrama de flujo del programa principal.
   - Cualquier detalle sobre la implementación que se considere relevante.
8. Se entrega: el fichero .cpp, un fichero texto con la información de un laberinto, varios videos con la ejecución del programa donde se muestre la generación y resolución de un laberinto, la memoria explicativa y la documentación generada con Doxygen en html.
   La entrega del código se realizará por medio de un repositorio de control de versiones Git en el servidor `https://diversion.uv.es`. La memoria explicativa, los posibles videos y la documentación en html se entregarán a través de Aula Virtual.
9. El repositorio para cada alumno con el esqueleto del fichero .cpp, ficheros de ejemplo y videos está indicado en los ficheros reporsitorios Lx.pdf.
10. Se debe seguir la guía de Estilo de C++ (ver pdf en Aula Virtual). El programa debe estar convenientemente documentado utilizando la herramienta Doxygen.

Procedimiento de Entrega:

Durante el desarrollo del proyecto se deben seguir las recomendaciones y forma de trabajo propuestas durante la sesión de control de versiones de la asignatura ISU. Se tendrá en cuenta en la calificación final del proyecto.

Es necesario realizar un *commit* cada vez que se añada una nueva estructura de datos, una función, un fichero o cuando se complete una funcionalidad o en general cualquier cambio que se considere relevante. Los mensajes de los *commit* deben explicar qué se ha completado, evitando mensajes genéricos. Siempre que se finalice una tarea del enunciado debe quedar claramente reflejado en el mensaje de *commit*.

El profesor os hará llegar el subgrupo al que pertenecéis cada pareja. Se han habilitado permisos al profesor de laboratorio. **Nuestros usuarios son:** `elenad, magamon y mperezm.`

Parte 1: Generación de un Laberinto

En este proyecto se implementará un método sencillo para la generación y resolución de laberintos. Vamos a trabajar con laberintos perfectos de tamaño relativamente pequeño con forma de cuadrado. Un **laberinto se dice que es perfecto** si cumple las siguientes condiciones:

i. Cada punto es alcanzable desde la entrada (no existen zonas desconectadas).
ii. Existe un solo camino entre dos puntos.
iii. No tiene bucles. Esto implica que no existen islas.
iv. Tiene un único punto de entrada y un único punto de salida.

Este tipo de laberintos también son conocidos como **laberintos simplemente conectados**. Se asume que la entrada y la salida se encuentran en las paredes exteriores. En concreto, la entrada se encuentra en la esquina inferior izquierda y la salida en la esquina superior derecha. La Figura 1 muestra dos ejemplos con diferentes tamaños.

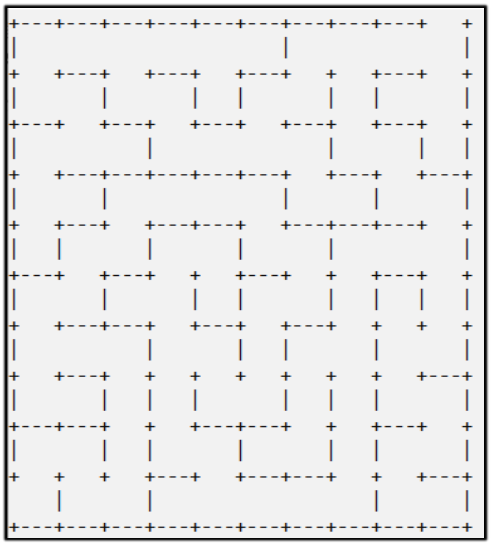


Figura 1a: Laberinto de tamaño 10 x 10 celdas.

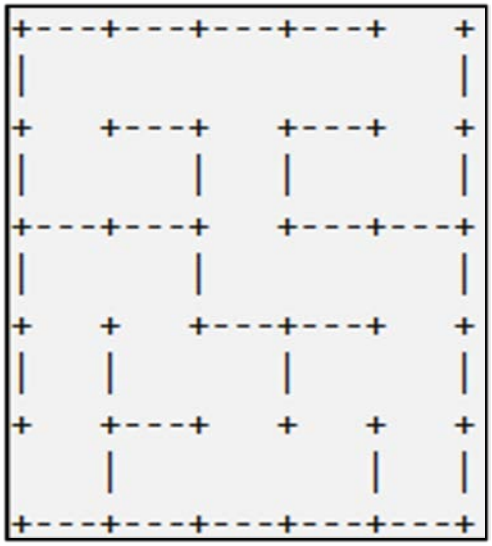


Figura 1b: Laberinto de tamaño 5 x 5 celdas.

Para construir un laberinto se parte de una retícula (vértices de la rejilla en naranja o celdas en azul) y seguidamente se conecta cada celda con sus cuatro vecinas tal y como se muestra en la Figura 2.

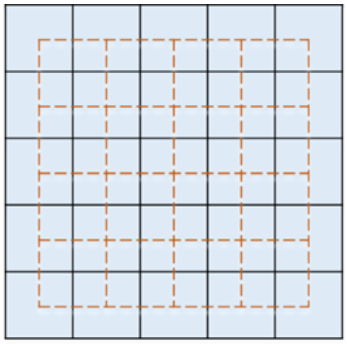


Figura 2: Retícula de tamaño 5x5.

Los nodos corresponden con los centros de las cuadriculas en azul y las paredes o muros entre nodos se muestran en líneas de color negro.

Una celda de coordenadas `(i,j)` posee cuatro vecinas: al Norte, al Sur, al Este y al Oeste. Vienen definidas por los pares de índices siguientes:

```
(i-1, j)     Celda al Norte de (i, j)
(i, j+1)     Celda al Este de (i, j)
(i+1, j)     Celda al Sur de (i, j)
(i, j-1)     Celda al Oeste de (i, j)
```

Las cuatro celdas vecinas se distribuyen de la forma siguiente alrededor de la celda central:

	(i-1,j)	
(i,j-1)	(i,j)	(i,j+1)
	(i+1,j)	

Es muy importante tener en cuenta el **efecto de borde** y elegir de forma adecuada las celdas vecinas para no salirse del laberinto. Concretamente, las celdas en el borde izquierdo no tienen vecinas al Oeste, las celdas en el borde superior no tienen vecinas al Norte, las celdas en el borde derecho no tienen vecinas al Este y finalmente las celdas del borde inferior no tienen vecinas al Sur.

Una forma de generar el laberinto es aplicando un algoritmo recursivo aleatorio basado en una búsqueda en profundidad. Es uno de los más utilizados gracias a la simplicidad del planteamiento, sencillez a la hora de la implementación y rapidez de ejecución.

El algoritmo comienza en una celda fija que se marca como visitada, en esta implementación se elige la celda de entrada (esquina inferior izquierda). A continuación, se selecciona de forma aleatoria una celda vecina que no haya sido hasta el momento visitada, se etiqueta como visitada y se elimina la pared entre ellas. Ésta celda pasa a ser la celda actual y se continua. Si la celda actual no tiene vecinas sin visitar, se vuelve hacia atrás hasta la última celda que tenía vecinas sin visitar. El proceso termina cuando no quedan vecinas por visitar. A esta técnica se le conoce como **retroceso** o vuelta atrás (*backtracking* en inglés, `https://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s`).

En el **Video** `Generacion1.mp4` y **Video** `Generacion2.mp4` se visualiza la construcción de un laberinto detenidamente y se aprecia el proceso de retroceso. Observa conforme cada vez hay más celdas visitadas, los saltos son más grandes cuando se produce la vuelta atrás.

Una descripción en pseudocódigo del algoritmo es:

```
Algoritmo Crear Laberinto
función CrearLabetinto ()
  celdaActual ← reticula(0,0)
  Crear (celdaActual)
fin_funcion

funcion Crear (celda)
  marcar celda como visitada
  mientras quedan vecinas de celda no visitadas hacer
     siguienteCelda ← elegir aleatoriamente una vecina de celda no visitada
     eliminar la pared entre celda y siguienteCelda
     Crear (siguienteCelda)
  fin_mientras
fin_funcion
```

Al tratarse de un algoritmo recursivo, su implementación puede provocar un desbordamiento de la pila de llamadas a funciones si el laberinto es de gran tamaño. En este trabajo se van a construir laberintos pequeños en forma de cuadrado de tamaño 10 x 10 celdas.

La Figura 1 muestra dos ejemplos de la construcción de un laberinto aplicando el algoritmo anterior. Es importante elegir aleatoriamente entre las celdas vecinas pendientes de visitar. De lo contrario, se generan patrones con caminos en orientaciones fijas. Por ejemplo en la Figura 3a se muestra un laberinto que se ha implementado usando el orden de visita de los nodos vecinos Este, Norte, Oeste y Sur (“ENOS”) y en la Figura 3b se ha elegido el orden Sur, Este, Norte y Oeste (“SENO”). Su aspecto es artificial y bastante diferente a los de la Figura 1. Ver **Video** OSEN.mp4

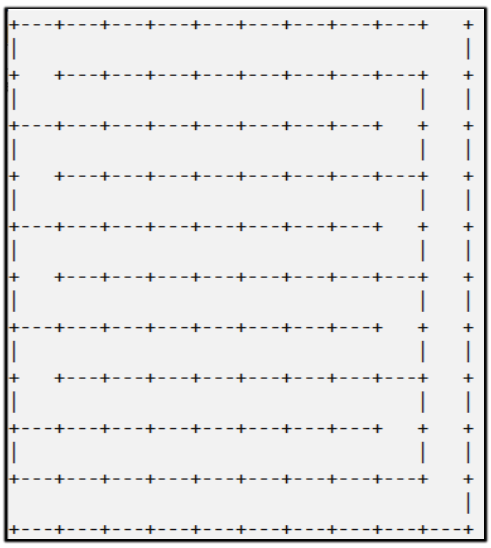


Figura 3a: Orden de visita “ENOS”.

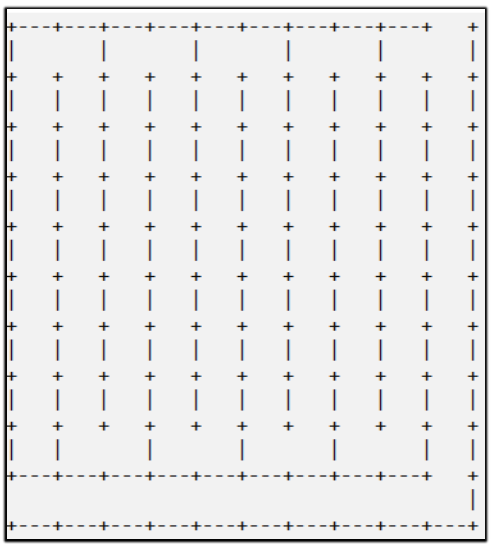


Figura 3b: Orden de visita “SENO”.

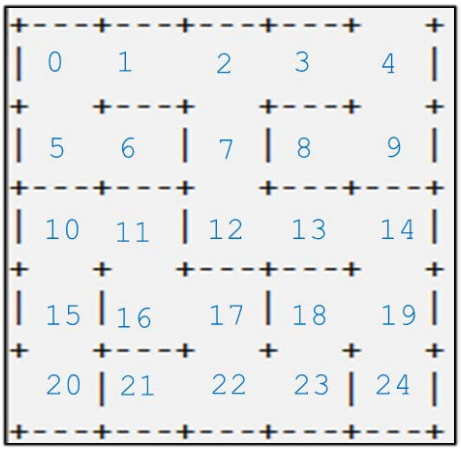
Para la implementación se van a utilizar dos matrices: una para almacenar la información de los nodos y otra para almacenar las paredes del laberinto, es decir, las adyacencias entre nodos al haber eliminado la pared entre ellos. A ésta última matriz también se le denomina **matriz de adyacencias**. El laberinto está completamente determinado por los valores de esta matriz.

En concreto, para cada celda se guarda si ya ha sido o no visitada y los identificadores de los cuatro vecinos (se usan en las funciones de generación y resolución). En la matriz de adyacencias se guarda para cada nodo si es adyacente o no a otro, es decir, si se ha eliminado la pared que los separa. En consecuencia, se trata de una matriz simétrica.

El identificador de una celda de coordenadas `(i,j)` viene dado por el valor `i * TAM + j`. Los identificadores para sus vecinos al Norte, Este, Oeste y Sur son respectivamente:

```
idN = (i - 1) * TAM + j;
idE = i * TAM + j + 1;
idO = i * TAM + j - 1;
idS = (i + 1) * TAM + j;
```

Debido a que el tamaño del laberinto es finito, en las celdas del borde no existen algunos nodos vecinos, en este caso al identificador se le asigna el valor -1. La Figura 4 muestra las etiquetas correspondientes a las celdas en un laberinto de 5 x 5.




```
a[1][2] = a[2][1] = true
a[1][6] = a[6][1] = false

a[0][1] = a[1][0] = true
a[0][5] = a[5][0] = true
```

Figura 4: Celdas con sus identificadores. Algunos valores de adyacencias.

En este ejemplo se cumple que la celda etiquetada con el identificador 0 es adyacente a las celdas con identificadores 1 y 5 porque no existen muros o paredes entre ellas. Por tanto, la matriz de adyacencias `a[0][1] = a[1][0] = true` y `a[0][5] = a[5][0] = true`. Sin embargo, la celda etiquetada con el identificador 1 es adyacente a la 0 y a la 2 pero no a la celda 6, `a[1][2] = a[2][1] = true` y `a[1][6] = a[6][1] = false`.

Se deben definir los siguientes tipos de datos e implementar las funciones que se detallan a continuación:

```
const int TAM = 10;

struct Nodo
{
   bool v;
   int idN, idE, idS, idO;
};

typedef Nodo MNodos [TAM][TAM];

typedef bool MParedes [TAM * TAM][TAM * TAM];

void Inicializar (MNodos, MParedes);

void Crear (unsigned int, MNodos, MParedes);

bool NoVisitada (unsigned int, MNodos, MParedes, unsigned int &);

void Mostrar (MParedes);
```

Parte 2: Resolución de un Laberinto

Una forma sencilla de resolver un laberinto perfecto es usar la regla de la mano derecha si eres diestro (o de la mano izquierda si eres zurdo). La idea de forma esquemática se puede expresar como:

- Siempre gira a tu derecha, si puedes.
- Si no puedes girar a la derecha, avanza hacia delante.
- Si no puedes girar a tu derecha, ni avanzar hacia delante, gira a tu izquierda.
- Si no puedes girar a tu derecha, ni avanzar hacia delante, ni girar a tu izquierda, date la vuelta porque has llegado a un punto sin salida.

Aplicando esta lógica encontrar la salida del laberinto está garantizado. **Ver Videos** Resolver1.mp4, Resolver2.mp4, Resolver3.mp4.

A continuación, se indica el pseudocódigo de este método:

```
Algoritmo Resolver Laberinto Regla Mano Derecha

función ResolverLabetinto ()
   estado ←  posición de la entrada y mira al Norte
   Resolver (estado)
fin_funcion

funcion Resolver (estado)
   mientras no estemos en la celda de salida hacer
      si mira al Norte entonces EjecutarMov (estado, "ENOS");
      si mira al Oeste entonces EjecutarMov (estado, "NOSE");
      si mira al Sur entonces EjecutarMov (estado, "OSEN”);
      si mira al Este entonces EjecutarMov (estado, "SENO");
   fin_mientras
fin_funcion
```

```
funcion EjecutarMov (ref: estado, orden)
    i  ← 1
   mientras no posible giro en orden(i) hacer
       i  ← i + 1
   fin_mientras
   modificar la orientación de mira y filas o columnas del estado
fin_funcion
```

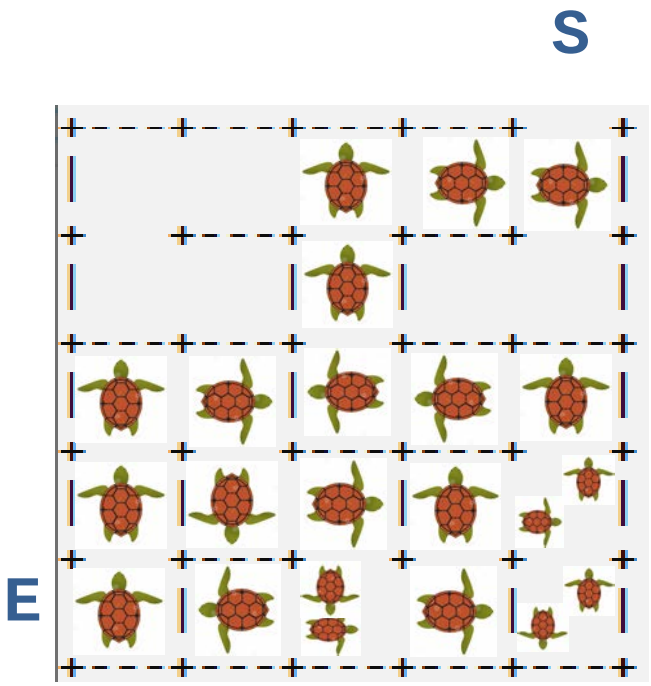


Figura 5: Recorrido desde entrada a la salida.
Durante el trayecto la tortuga encuentra dos “callejones sin salida”.

Ten en cuenta que el método implica que el orden de evaluación de los posibles movimientos depende de la orientación actual. En particular:

- Si se mira hacia el Norte, el orden de evaluación es Este, Norte, Oeste, Sur.
- Si se mira hacia el Oeste, el orden de evaluación es Norte, Oeste, Sur, Este.
- Si se mira hacia el Sur, el orden de evaluación es Oeste, Sur, Este, Norte.
- Si se mira hacia el Este, el orden de evaluación es Sur, Este, Norte, Oeste.

En la Figura 5 se muestra el recorrido que realiza una tortuga hasta llegar a la salida: "despacio pero segura". Durante el camino encuentra dos puntos muertos. En esas configuraciones es necesario dar la vuelta.

Se deben definir los siguientes tipos de datos e implementar las siguientes funciones:

```
struct Estado
{
   unsigned int f; // fila de posición actual
   unsigned int c; // columna de posición actual
   char mira; // al N, E, S, O
};

void EjecutarMov (MNodos, MParedes, Estado &, string);

void Resolver (Estado &, MNodos, MParedes);

char Menu();

void Leer (ifstream &, MParedes);

void Guardar (ofstream &, MParedes);

void IrA (unsigned int, unsigned int);

string AleatorizarDir ();

void MostrarEstado (Estado);
```

Una ejecución completa del programa usando el menú se ilustra en el **Video** `Ejecucion.mp4`.

Observaciones:

- ✓ Es recomendable implementar las siguientes funciones auxiliares:
  - Una función para aleatorizar las orientaciones de búsqueda de las vecinas aún no visitadas para la generación del laberinto.
  - Una función para mostrar la posición actual y la orientación sobre la retícula.
  - Una función para encontrar la posición donde colocar el cursor en la pantalla dada la fila y columna de la matriz.

- ✓ Para visualizar el proceso de la generación y resolución es conveniente detener la ejecución durante una cantidad fija de milisegundos. Para ello puedes usar la función `sleep_for()` que se encuentra en las librerías `<thread>` y `<chrono>`. Añade la opción `-std=c++11` cuando se llame al compilador.

```
this_thread::sleep_for (chrono::milliseconds(cantidad_de_ms));
```

- ✓ La semilla para generar números aleatorios se inicializa en la función `Inicializar()`.
- ✓ Presta atención especial para no salirse del laberinto en las funciones: `NoVisitada()`, `EjecutarMov()`.
- ✓ El laberinto está completamente determinado por los valores de la matriz de adyacencias, por tanto, las funciones leer desde fichero y guardar en fichero solo necesitan está información.
- ✓ Material que se acompaña en https://diversion.uv.es

  - `Fichero laberinto_alumno.cpp`
  - `Videos: Generacion1.mp4, Generacion2.mp4, OSEN.mp4, Resolucion1.mp4, Resolucion2.mp4, Resolucion3.mp4, Ejecucion.mp4`
  - `Fichero lab.txt`

La función principal es:

```
int main()
{
   int main ()
{
   HANDLE h = GetStdHandle (STD_OUTPUT_HANDLE);
   CONSOLE_CURSOR_INFO cursor;
   cursor.bVisible = FALSE;
   SetConsoleCursorInfo (h, &cursor);
   system("MODE CON COLS=42 LINES=22");

   MNodos n;
   MParedes v;
   char op;
   string nombre;
   ifstream f_in;
   ofstream f_out;

   do
   {
      op = Menu();
      switch (op)
      {
         case 'A':
            Inicializar (n, v);
            Crear ((TAM - 1) * TAM, n, v, h); //id de la Entrada
            Mostrar (v, h);
            break;
         case 'B':
            cout << "Introduce el nombre: ";
            cin >> nombre;
            f_in.open (nombre.c_str());
            if (!f_in) cout << "Error al abrir el fichero " << nombre;
            else
            {
               Leer (f_in, v);
               f_in.close ();
            }
            break;
```

```
         case 'C':
            cout << "Introduce el nombre: ";
            cin >> nombre;
            f_out.open (nombre.c_str());
            if (!f_out) cout << "Error al abrir el fichero " << nombre;
            else
            {
               Guardar (f_out, v);
               f_out.close ();
            }
            break;
         case 'D':
            Mostrar (v, h);
            break;
         case 'E':
            {
               Estado e = {TAM - 1, 0, 'N'}; //posicion actual en la E
               Resolver (e, n, v, h);  //siempre encuentra salida
               IrA (TAM, 0, h);
            }
      }
      system("pause");
   }
   while (op != 'F');

   return 0;
}
```