Práctica 5: Programas C++ con registros, matrices y matrices de registros

5.1. Objetivos de la práctica

En esta práctica se va a trabajar con vectores de registros (tarea 5.2), matrices (tarea 5.4) y matrices de registros (tarea 5.3). Además, se va a hacer énfasis en la metodología de diseño descendente, puesto que la descomposición modular de los distintos problemas no se ofrece de forma tan pautada como en prácticas anteriores.

Los problemas planteados en **las tareas relativas a vectores de registros y matrices de registros (tareas 5.2 y 5.3) forman parte de la entrega obligatoria de esta práctica**, aunque se recomienda resolver también el problema relativo a matrices (tarea 5.4), ya que contribuirá a la calificación del apartado «Pruebas con carácter voluntario» establecido en la guía docente de la asignatura.

Al igual que en prácticas anteriores, a la sesión asociada a esta práctica hay que acudir con el trabajo que se describe a continuación razonablemente avanzado, con objeto de aprovecharla con un doble objetivo:

- Consultar al profesorado las dudas surgidas y las dificultades que hayan impedido resolver satisfactoriamente los problemas planteados y, con su ayuda, tratar de aclarar ideas y avanzar en la resolución de los problemas surgidos.
- Presentar al profesorado el trabajo realizado para que este lo pueda revisar, advertir de posibles errores y defectos y dar indicaciones sobre cómo mejorarlo y, en su caso, corregirlo.

En el repositorio `https://github.com/prog1-eina/practica5` tienes el área de trabajo para esta práctica, con la estructura de directorios y ficheros necesaria para que el fichero «`Makefile`» y las tareas de compilación, ejecución y depuración de Visual Studio Code funcionen adecuadamente.

Puedes descargarte el área de trabajo completa (botón Code › Download ZIP de la web del repositorio), y descomprimirla en tu directorio «`programacion1`» como «`practica5`» (como siempre, recuerda borrar el sufijo «`-main`» que añade GitHub al preparar el fichero comprimido para su descarga).

Tarea 5.2 Números complejos

Un número complejo es un número que puede ser expresado de la forma $a + bi$, donde a y b son números reales e i es la solución a la ecuación $x^2 = -1$. Como ningún número real satisface esta ecuación, a i se le llama unidad imaginaria. Dado el número complejo $a + bi$, se denomina parte real a a y parte imaginaria a b (puedes leer más sobre los números complejos en la Wikipedia[1]).

Como primera tarea en esta práctica, **escribe un programa que genere un vector de 6 números complejos, con sus partes real e imaginaria elegidas en un intervalo (x, y) de manera aleatoria**. El programa debe, en primer lugar, solicitar los límites de este intervalo al usuario, garantizando que los valores introducidos por el usuario cumplen que $x < y$. Después, mostrará al usuario todos los números complejos generados. Posteriormente, mostrará al usuario el número complejo de mayor módulo y el número complejo de menor módulo. Recuerda que el módulo de un número complejo $a + bi$ se calcula como $\sqrt{a^2 + b^2}$. El programa finalizará mostrando la suma de todos los números complejos generados. Recuerda también que la suma de dos números complejos $(a + bi)$ y $(c + di)$ es $(a + c) + (b + d)i$. Todos los números decimales se mostrarán con una precisión de **dos dígitos**.

Para escribir este programa, **deberás definir un tipo de datos que permita representar un número complejo, utilizar un vector de 6 componentes del tipo de datos que has definido y realizar un diseño modular con, al menos, procedimientos o funciones para calcular lo siguiente**:

- Escribir un número complejo en la pantalla.
- Obtener las posiciones de un vector números complejos donde se ubican aquellos que tienen mayor y menor módulo.
- La suma de todos los números complejos contenidos en un vector.

Ejemplo de interacción:

```
Introduce los límites del intervalo (x < y): 3 -2
Introduce los límites del intervalo (x < y): 10.8 3
Introduce los límites del intervalo (x < y): -3.14   2

Números complejos generados:
1.76+0.15i, -0.47-0.03i, 0.00-1.28i, -0.50+0.00i, -1.99+0.15i, -1.94+1.36i

Números complejos de mayor y menor módulo, respectivamente:
-1.94+1.36i y -0.47-0.03i

Suma:
-3.14+0.36i
```

Escribe el código de este programa en el fichero «`complejos.cpp`» del directorio «`src`» del repositorio «`practica5`» que te has descargado de GitHub.

Recuerda actualizar la cabecera del fichero y especificar todas las funciones que escribas.

[1] Wikipedia contributors, «Complex number», *Wikipedia, The Free Encyclopedia,* `https://en.wikipedia.org/w/index.php?title=Complex_number` (accedido el 9 de noviembre de 2023).

Generación de números aleatorios

En informática, los números puramente aleatorios no existen, y por eso se les llama *números pseudoaleatorios*. Para generar números pseudoaleatorios en C++ puede utilizarse la función `rand()`[2] de la biblioteca `<cstdlib>`, que genera números **enteros** pseudoaleatorios entre 0 y `RAND_MAX`. Por ejemplo, el siguiente fragmento de código escribirá en la pantalla siete números pseudoaleatorios:

```
for (unsigned i = 0; i < 7; i++) {
    cout << rand() << endl;
}
```

Si ejecutas el código anterior varias veces, comprobarás que la secuencia de números generada es siempre la misma. Esto es así porque los números pseudoaleatorios se calculan a través de funciones matemáticas que permiten ir obteniendo una secuencia determinada de números, pero de aparencia aleatoria. Cada vez que se llama a `rand()`, se obtiene el siguiente número de esa secuencia, usando para el cálculo el último número pseudoaleatorio que se había calculado anteriormente. En el programa que se pide desarrollar, que la secuencia pseudoaleatoria sea siempre la misma, lejos de suponer un problema, puede suponer en realidad una ventaja: mientras estés desarrollando el programa, distintas ejecuciones de tu programa utilizarán la misma secuencia pseudoaleatoria, lo que te facilitará comprobar la corrección de los resultados y la corrección de errores.

Sin embargo, esta situación no es deseable en entornos reales. Si se desea que la secuencia de números pseudoaleatorios sea distinta cada vez, es preciso invocar a la función `srand()`[3] (normalmente una única vez al principio del programa) antes de la primera invocación a `rand()`. La función `srand()` permite establecer la *semilla* inicial, que es el primer número que se usará para generar la secuencia de números pseudoaleatorios. Una determinada semilla genera siempre la misma secuencia de números pseudoaleatorios, mientras que las secuencias asociadas a semillas distintas son también distintas.

La implementación de la función `rand()` es dependiente de la implementación. Por tanto, la calidad de la secuencia de números pseudoaleatorios que produce, así como su distribución y periodo, no están garantizadas. C++ dispone de una biblioteca predefina denominada `<random>`, que es la que debería utilizarse en programas en los que la calidad de la pseudoaleatoriedad sea importante (no es el caso del programa que se solicita en esta práctica).

Tarea 5.3 Modificación de imágenes de mapas de bits (BMP)

En informática, una forma de representar imágenes es a través de *mapas de bits*: la imagen se representa a través de una matriz de puntos discretos, denominados *píxeles* (del acrónimo en inglés *picture element*). Cada píxel representa la menor unidad en la que se puede descomponer la imagen y tiene un color determinado y homogéneo. Este color se representa habitualmente a través de una combinación de diferentes intensidades de luz roja, verde y azul. Así, por ejemplo:

- un píxel de color blanco se representa a través de la combinación de intensidades máximas de luz roja, verde y azul;
- un píxel negro, a través de la combinación de intensidades nulas de luz roja, verde y azul;
- un píxel amarillo a través de la combinación de intensidades máximas de luz roja y verde, y nula de luz azul; y

[2] Descripción de la función en `http://www.cplusplus.com/reference/cstdlib/rand/`.
[3] Descripción de la función en `http://www.cplusplus.com/reference/cstdlib/srand/`.

- un píxel gris a través de una combinación de intensidades intermedias de luz roja, verde y azul.

Habitualmente, el nivel de intensidad de cada canal de luz se representa a través de un entero positivo entre 0 y 255, donde 0 representa una intensidad nula y 255, la intensidad máxima.

El formato *Windows bitmap* (BMP) es un formato de fichero para almacenar imágenes de mapas de bits, utilizado sobre todo en el sistema operativo Windows. En estos ficheros se almacenan, entre otros datos, la matriz de píxeles que forman la imagen en sí (que se almacenan por filas), así como el número de píxeles de cada fila (altura de la imagen) y de cada columna (anchura de la misma).

En esta tarea, vamos a trabajar con ficheros BMP de 24 bits de hasta 8192 × 8192 píxeles. Por las características concretas de este formato, tanto la altura como la anchura de las imágenes serán múltiplos de 4.

En el módulo «`imagen-bmp`» os facilitamos las definiciones de dos tipos registro denominados `Pixel` e `Imagen`. El tipo `Pixel` permite representar píxeles a través de tres campos denominados `rojo`, `verde` y `azul`. Como solo es necesario representar enteros entre 0 y 255, utilizamos el tipo **`char`**, que se codifica con 8 bits (aunque no nos interesan sus valores como carácter, sino sus valores enteros).

El tipo `Imagen` permite representar el contenido completo de un fichero de imagen BMP: su anchura, altura y la matriz de píxeles. Se incluyen también dos campos adicionales que representan la información de la cabecera del fichero BMP dividida en dos partes para tener una representación completa del contenido del fichero en el fichero. Aunque no vas a tener que trabajar con los datos de estos campos `cabeceraParte1` y `cabeceraParte2` en el programa que se te solicita escribir, en las funciones de lectura y escritura que se describen a continuación sí que hemos tenido que hacer uso de ellos.

En el mismo módulo se proporcionan dos funciones que permiten leer un fichero de tipo BMP (procedimiento `leerImagen()`) y almacenar su contenido en un registro `Imagen` y almacenar el contenido de un registro `Imagen` en un fichero con formato BMP (procedimiento `guardarImagen()`). El código de estas funciones lo utilizaremos de ejemplo en clase de teoría más adelante en el tema dedicado a ficheros binarios.

En el fichero «`imagen-bmp-main.cpp`» es donde escribirás el código correspondiente al problema que se plantea a continuación. El fichero que te has descargado de GitHub contiene un ejemplo en el que se utilizan las funciones de lectura y escritura de imágenes mencionadas anteriormente. Ejecútalo para comprobar que todo funciona correctamente. Si es así, verás como se genera en la carpeta «`data`» una copia del fichero «`prog1.bmp`» denominada «`imagen-generada.bmp`».

Observa que en la función `main` de este fichero se ha declarado una variable de tipo `Imagen` como *estática*, pese a que se trata de un concepto con el que no se ha trabajado en el curso. La razón de su utilización reside en el hecho de que la variable `img` ocupa 192 megabytes y es demasiado grande para ser almacenada en la pila (el segmento de memoria donde se ubican habitualmente las variables locales de una función o procedimiento). Con el modificador **`static`** se indica al compilador que la ubique en otro segmento de memoria (concretamente, en el segmento de datos). Conocerás más detalles sobre los distintos segmentos de memoria en la asignatura Arquitectura y Organización de Computadores 1 del 2.º cuatrimestre.

Se pide un programa que tenga el siguiente comportamiento:

- El programa pide el nombre de un fichero con formato BMP. Como los ejemplos que os hemos pasado están en una carpeta denominada «`data`», si el usuario quiere utilizar uno de ellos, su respuesta debe comenzar con la cadena "data/". **No es responsabilidad del programa comprobar este aspecto.**
- Si el fichero puede leerse correctamente, el programa solicita una opción a través de un carácter:

‘N’ para negativizar (invertir los colores de una imagen), ‘P’ para permutar los canales de los píxeles y ‘R’ para girar la imagen 180°. Debe ser indiferente que el usuario escriba la opción en mayúsculas o minúsculas. Encontrarás a continuación una explicación de qué tenemos que hacer para negativizar, permutar o rotar una imagen.

- El programa pide el nombre del fichero en el que guardar la imagen transformada y finalmente la guarda.

Para negativizar una imagen, hay que invertir la intensidad de cada canal de luz de cada píxel de la imagen. Para ello, si x es la intensidad de uno de los canales (rojo, verde o azul) expresada como un entero entre 0 y 255, la expresión $255 - x$ representa la intensidad opuesta.

Por ejemplo, si la imagen original fuese la del fichero «`datos/unizar.png`», la imagen negativizada sería la siguiente:

Para permutar los canales de una imagen, hay que intercambiar los valores de cada canal de luz de cada imagen, de forma que el nuevo valor del canal rojo sea el antiguo valor del canal verde; el nuevo valor del canal verde, el antiguo del canal azul y el nuevo valor del canal azul, el que tenía anteriormente en el rojo.

Por ejemplo, si la imagen original fuese la del fichero «`data/prog1.png`», la imagen permutada sería la siguiente:

Para girar una imagen 180º, basta con cambiar los datos de la matriz $ancho \times alto$ donde se guardan los píxeles: el elemento (i, j) de la matriz pasará a ser el elemento $(alto - 1 - i, ancho - 1 - j)$.

Por ejemplo, si la imagen original fuese la del fichero «`data/prog1.png`», la imagen rotada sería la siguiente:

Se muestran a continuación cuatro ejemplos independientes de ejecución del programa:

```
Escriba el nombre de un fichero BMP: data/prog1.bmp
Imagen "data/prog1.bmp" leída con éxito.
Escriba una opción (N - negativizar; P - permutar; R - rotar): N
Escriba el nombre del fichero destino: data/negativo.bmp
Imagen "data/negativo.bmp" creada con éxito.
```

```
Escriba el nombre de un fichero BMP: data/unizar.bmp
Imagen "data/unizar.bmp" leída con éxito.
Escriba una opción (N - negativizar; P - permutar; R - rotar): r
Escriba el nombre del fichero destino: data/rotada.bmp
Imagen "data/rotada.bmp" creada con éxito.
```

```
Escriba el nombre de un fichero BMP: eina.bmp
No se ha encontrado el fichero "eina.bmp".
```

```
Escriba el nombre de un fichero BMP: data/ada.bmp
Imagen "data/ada.bmp" leída con éxito.
Escriba una opción (N - negativizar; P - permutar; R - rotar): J
Opción desconocida
```

Si quieres realizar pruebas de este programa con imágenes distintas a las que os hemos facilitado, puedes utilizar el programa Paint de Windows. Asegúrate de guardar la imagen como una imagen BMP (Archivo » Guardar como... » Imagen BMP » Mapa de bits de 24 bits (*.bmp;*.dib)) y de que la altura y la anchura sean ambas múltiplo de 4 y no superen el valor 8192.

En `https://cloud.unizar.es/index.php/s/gb8LegCzgSxWjFc` tienes disponible una imagen comprimida de 4096 × 2304 píxeles, de un tamaño sensiblemente mayor que las proporcionadas en el directorio «`data`» de la práctica. Se recomienda hacer pruebas también con ese fichero.

A modo de referencia, se muestra la imagen original y las resultantes de negativizar, permutar y rotar:

Imagen «`ada.bmp`» original

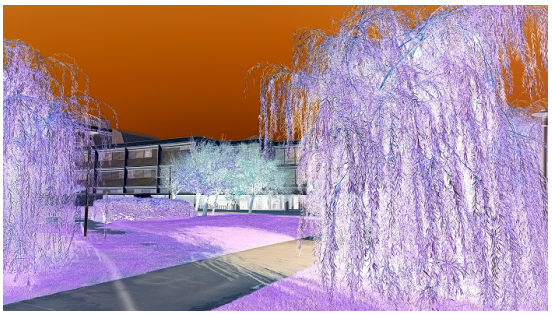

Imagen «`ada.bmp`» negativizada

Imagen «`ada.bmp`» permutada

Imagen «`ada.bmp`» rotada

Tarea 5.4 El Juego de la vida

El *Juego de la vida* es un autómata celular diseñado por el matemático británico John Horton Conway (1937–2020) que simula la evolución de una colonia de seres unicelulares sobre las celdas de un tablero rectangular. Cada celda puede estar vacía o habitada por una sola célula viva. Cada celda tiene ocho celdas vecinas (sus celdas adyacentes vertical, horizontal y diagonalmente), excepto si está ubicada en la periferia del tablero.

Cada cierto tiempo se produce una renovación generacional siguiendo las siguientes reglas:

- Una célula viva muere de inanición si en las celdas vecinas a la que ocupa hay menos de dos células vivas.
- Una célula viva sobrevive si en las celdas vecinas a la que ocupa hay 2 o 3 células vivas.
- Una célula viva muere por superpoblación si en las celdas vecinas a la que ocupa hay más de tres células vivas.
- En una celda vacía nace, por reproducción, una célula viva si en sus celdas vecinas hay exactamente tres células vivas.

Pese a que el juego original diseñado por John H. Conway tiene lugar en un tablero infinito, vamos a limitarnos a trabajar en un tablero de dimensiones máximas `MAX_FILAS`×`MAX_COLUMNAS`, siendo `MAX_FILAS` y `MAX_COLUMNAS` dos constantes definidas en el fichero «`juego-vida.cpp`» suministrado.

5.4.1. Trabajo a desarrollar en esta tarea

En esta tarea te pedimos **que escribas un programa que implemente el *Juego de la vida* a partir de una configuración inicial parcialmente suministrada por el usuario**. Este programa tiene que tener el siguiente comportamiento:

- El programa comienza solicitando al usuario el número de filas del tablero con el que desea trabajar. La solicitud se repite hasta que el usuario introduce un entero mayor estricto que 0 y menor o igual que `MAX_FILAS`.
- El programa solicita a continuación el número de columnas del tablero con el que desea trabajar. La solicitud se repite hasta que el usuario introduce un entero mayor estricto que 0 y menor o igual que `MAX_COLUMNAS`.
- El programa pide el número de generaciones a simular. La solicitud se repite hasta que el número introducido es mayor o igual que 0.
- A continuación, el programa inicializa las filas y columnas indicadas por el usuario de un tablero de forma aleatoria, de forma que en cada celda haya una célula con una probabilidad del 20 %.
- El programa debe escribir en la pantalla el estado inicial de ese tablero. Para ello, escribirá en la pantalla el mensaje "Generación 0" y, seguidamente, a partir de la línea siguiente, mostrará el tablero, a razón de una fila del mismo por línea. Las celdas vacías se representarán con un espacio en blanco y las celdas con una célula viva, con un asterisco (carácter '$*$'). Para que visualmente resulte más atractivo, cada una de las columnas de una celda se separará de la siguiente con un espacio en blanco.
- El programa continuará escribiendo en la pantalla los estados que va alcanzando el juego conforme pasa el tipo y se aplican las reglas de supervivencia descritas anteriormente. Para ello, escribirá el mensaje "Generación *i*", donde *i* es el número de iteración y, a continuación, el estado del tablero según lo enunciado en el punto anterior.

  En el fichero se proporciona el código de dos funciones que, utilizando *secuencias de escape*[4], pueden servir para mejorar la apariencia gráfica de la simulación. Se sugiere utilizar antes de mostrar la generación inicial la función `borrarPantalla` una única vez, para borrar el texto escrito en el terminal y mover el cursor a la primera columna de la primera línea. La función `subirCursor` sube la posición del cursor el número de líneas determinado por su parámetro y se puede utilizar antes de escribir cada generación en la pantalla, para sobrescribir la escritura de la generación anterior.

  Si estas funciones no se ejecutaran correctamente en tu terminal porque este no soporta correctamente la secuencias de escape, simplemente no las utilices.
- El programa cesará su ejecución cuando se hayan mostrado en la pantalla tantos estados como generaciones haya indicado inicialmente el usuario o cuando se alcance un estado en el que no quede ninguna célula superviviente.

  Si la simulación termina porque todas las células de la colonia han muerto, el programa debe indicarlo escribiendo en la pantalla el mensaje "Colonia extinguida.". En caso contrario, debe escribir un mensaje indicando en número de células que quedan en la colonia.

A continuación se presenta un ejemplo de ejecución del programa:

[4] `https://en.wikipedia.org/wiki/ANSI_escape_code`


```
Número de filas: -1
Número de filas: 80
Número de filas: 10
Número de columnas: -10
Número de columnas: 120
Número de columnas: 30
Número de generaciones: -6
Número de generaciones: 20

Generación 0
 *   *               *         * *   * *     * * *
 *       *                     *                           *
   *   * *                               *             *
         *     *     *   *         *     *       * *     *
                 *     *       *
 *           *     *   *       *   * *           *
   * * *       *   * *         *                   *
 *   *       *   *     *
 *         *           *           *           *           *
 *       *   * *                 *                   * *
...

Generación 20


                   *
                 * * * *
     *         *   * * * *
   *     *       *     * * *     * *
   *       *     *   * * *   *     * *
   *     *           *   *                           * *
 * *   * *                                           * *
   *   *

Sobreviven 41 células.
```

5.4.2. Algunas notas sobre esta tarea

Utiliza una metodología de diseño descendente para estructurar el código en distintas funciones de no muchas líneas cada una y con un objetivo claro. A modo de ejemplo, se sugieren las siguientes:

- Una función que interactúe con el usuario pidiéndole los datos iniciales de configuración (dimensiones del tablero y número de generaciones).
- Una función que inicialice aleatoriamente el tablero del juego.
- Una función que escriba en la pantalla el estado del tablero en una determinada generación.
- Una función que cuente el número de células vivas en el tablero.

- Una función que determine si unas determinadas coordenadas del tablero (existentes o no) hay una célula o no. La pista 2 (explicada a continuación) da más información sobre esta posible función.
- Una función que, dadas las coordenadas de una celda del tablero, informe sobre el número de células vecinas existentes.
- Una función que actualice el tablero de una generación a otra aplicando las reglas del juego enunciadas anteriormente.
- Una función que realice una simulación del juego de la vida durante un determinado número de generaciones, escribiendo en la pantalla el estado del tablero tras cada actualización entre generaciones.
- La función `main()` del programa, que resuelve, en pocas líneas de código, el problema planteado utilizando el resto de funciones que hayas definido.

Recuerda actualizar las cabeceras de los ficheros y especificar todas las funciones que escribas.

Pistas

Pista 1: Al calcular la generación siguiente a una dada, todos los nacimientos y muerte de células deben ocurrir simultáneamente. Por ello, deben utilizarse dos tableros: uno contiene la generación actual y el otro, la siguiente generación.

Pista 2: Al calcular el número de células vivas vecinas a una celda dada, se puede evitar la complejidad de revisar los casos particulares correspondientes a las celdas periféricas (que tienen menos de 8 vecinas) utilizando una función que tenga el siguiente comportamiento: dado el tablero, sus dimensiones y las coordenadas (i, j) de una celda (coordenadas válidas o no), esta función debe devolver el valor booleano **`false`** en el caso de que las coordenadas no fueran válidas (es decir, cuando alguna de ellas, o ambas, sea menor que 0 o mayor que la dimensión máxima correspondiente). En caso contrario (es decir, cuando (i, j) representaran una celda válida del tablero), esta función devolverá **`true`** si la celda válida contiene una célula o **`false`** si está vacía.

5.5 Entrega de la práctica

La entrega deberá realizarse antes del **sábado 23 de noviembre a las 18:00**.

Antes de la fecha indicada, se deberán haber subido a Moodle los siguientes ficheros:

- «`complejos.cpp`»
- «`imagen-bmp-main.cpp`»

Si has realizado la tarea tarea 5.4 (*Juego de la vida*), puedes subir también el fichero «`juego-vida.cpp`».

Los ficheros «`imagen-bmp.cpp`» e «`imagen-bmp.hpp`» no hay que subirlos a Moodle, puesto que no deben ser modificados (ni añadiendo funciones en el mismo, ni modificando las cabeceras de las ya existentes). Si los modificas, es muy probable que el resultado de la corrección que realicemos los profesores del programa «`imagen-bmp-main.cpp`» termine en error de compilación, lo que acarreará una calificación de 0 en esa parte.

