

"Año del Bicentenario del Perú: 200 años de Independencia"

UNIVERSIDAD NACIONAL DE INGENIERÍA FACULTAD DE CIENCIAS



The game of Life

Docente:

Juan Carlos Espejo Delzo

Alumnos:

Gutiérrez Fernández, Bryan Antonio

Sección: B2

"(John Conway) Es uno de los más grandes matemáticos vivos, con un astuto sentido del humor, una curiosidad promiscua de un erudito y una compulsión por explicar todo sobre el mundo a todos los que lo habitan."

The Guardian (2015)

2020 – I



ÍNDICE

1. Introducción	2
2. Fundamento teórico	3
3. Planteamiento del problema	9
4. Solución del problema	12
5. Resultados	17
6. Bibliografía	20



1. INTRODUCCIÓN

1.1 OBJETIVOS

- Solucionar satisfactoriamente el problema asignado, utilizando como herramientas para tal fin, los temas y conceptos vistos a lo largo de todo el curso de Fundamentos de Programación.
- Crear un programa sencillo, fácil de entender y efectivo, que pueda ser utilizado para solucionar el problema planteado y pueda servir como ejemplo para el aprendizaje del lector.
- Desarrollar las habilidades teórico-prácticas de programación de los integrantes del equipo, por medio de la aplicación de todos los temas y conceptos del curso en un solo problema, integrándolos de manera efectiva.

1.2 ESTRUCTURA DE LA MEMORIA

Este informe está dividido en 6 capítulos o secciones, todos los cuales han sido cuidadosamente organizados para situar al lector en una determinada área del entendimiento del problema. Primero, en el capítulo 2 o Fundamento teórico, se le proporciona un breve “*background*” teórico al lector, necesario para situarlo en contexto de las herramientas que se utilizarán para solucionar el problema. Después, en el capítulo 3 o Planteamiento del problema, se presenta formalmente el problema a solucionar, además de analizar este con un enfoque sencillo que le permita al lector comprender sus aspectos más sustanciales, en suma, marcar las estrategias o “*insight*” para el desarrollo de la solución. A continuación, en el capítulo 4 o Solución del problema , se presenta el programa y se explica paso a paso la funcionalidad de cada aspecto de este, además de brindar un fundamento lógico para la utilización de las herramientas elegidas presentadas en el Fundamento teórico. Finalmente, en el capítulo 5 o Resultados, se muestra el funcionamiento del programa y se enseñan los productos que se han obtenido de la aplicación del programa.

Se le recomienda al lector, sobretodo si aún es un principiante, leer todos los capítulos del informe, y no solo la solución del problema, en principio porque cada capítulo aporta de manera distinta a su entendimiento, y en segundo porque se le quiere recordar al lector una máxima que todo buen programador siempre debe tener en cuenta: “resolver un problema no significa necesariamente haberlo *entendido*.”

Sin más, se procede con la presentación del informe.



2.FUNDAMENTO TEÓRICO

Antes de plantear formalmente el problema a solucionar, se introducirán brevemente conceptos clave para la satisfactoria comprensión y entendimiento del programa por parte del lector. Se advierte que, al ser este un trabajo cuyo principal objetivo es el de resolver un problema, no se profundizará en demasiado en las herramientas y la teoría detrás de ellas. Se asume entonces que el lector tiene el conocimiento mínimo necesario en computación y programación como para aventurarse a leer este trabajo. Se incluye el número de páginas del material que ha sido tomado como referencia con su respectiva etiqueta (revisar capítulo 6), en caso el lector quiera profundizar en algún tema de su interés. Todas las etiquetas sin número de página se refieren a la misma referencia inmediata anterior con número de página.

2.1 FUNCIONES DE USUARIO

Una función es una porción de código que, por su contenido, puede ser reutilizada en varias secciones de nuestro programa. Estas secciones de código reciben información (inputs) de otras funciones, como por ejemplo la función principal *main*, que procesan de acuerdo al contenido o instrucciones escritas en ellas, y finalmente, retornan un valor (por ejemplo, una función tipo *int* retorna un entero) o le “devuelven” el control (una función tipo *void* sólo ejecuta instrucciones sin retornar ningún valor) a la función que precisamente las llamó (ver pp. 155-160) [1]. A continuación, en la Figura 1, se detalla la sintaxis para la creación de una función de usuario.

C++ FUNCTION DEFINITION

FORM:

```
return_type function_name(parameter-declaration_list)
{
    statement_list
}
```

where:

return_type is the type of value the function returns or the keyword *void* if the function does not return a value;

function_name is an identifier that names the function;

parameter-declaration_list is a list, possibly empty, of comma-separated declarations of the function's parameters;

statement_list describes the behavior of the function.

Figura 1

Sintaxis de la definición de una función de usuario en C++[1].



De la Figura 1, podemos identificar 4 elementos de la función. Primero, *return_type*, que es el tipo de valor que efectivamente retornará la función, o, si no retorna nada, *void*; *function_name*, que es el identificador de la función, es decir su nombre; *parameter-declaration_list*, que es la lista de parámetros o inputs que recibirá nuestra función, si es que acaso los tiene (recordad el tipo *void*); y *statement_list*, que es la porción de código o lista de instrucciones con las que la función hará el trabajo requerido. Como regla general, se suele establecer una declaración de la función llamada prototipo antes de la función *main* para hacerle saber al compilador que la función existe y está siendo usada correctamente [1]. A continuación, se muestra la sintaxis para declarar un prototipo. Los elementos cumplen el mismo rol que en el de la sintaxis de la definición de una función.

C++ FUNCTION PROTOTYPE

FORM:

```
return_type function_name(parameter-declaration_list);
```

where

return_type, *function_name*, and *parameter-declaration_list* play the same roles as in a function definition. One difference is that parameter names are optional—only their types are required—but it is good practice to include them to indicate what the parameters represent.

Figura 2

Sintaxis de la declaración de un prototipo de función de usuario en C++[1].

2.2 RECURSIVIDAD

Una función se denomina recursiva cuando en su *statement_list* se llama a sí misma, es decir, es una función que se reutiliza dentro de su propio cuerpo. Esto se hace con la finalidad de evitar programar instrucciones redundantes y en algunos casos puede significar la diferencia entre un programa eficiente y uno que no lo es. Por lo general este tipo de funciones está compuesta de dos partes (ver pp. 379-380) [1]:

- a) Un caso base o “ancla”, que retorna un valor específico o finaliza (con la sentencia *return*) para una lista de parámetros previamente establecidos.
- b) Un paso inductivo o recursivo, en el que efectivamente se reutiliza la función.



2.3 ARREGLOS

Un arreglo es una colección de un tipo determinado de valores o caracteres almacenados de manera contigua. La sintaxis para la declaración de un arreglo unidimensional, esto es de una sola “fila”, se presenta a continuación.

ARRAY DECLARATION (SIMPLIFIED)

FORM:

```
type array_name [CAPACITY];
```

where:

type is any defined type (predefined or programmer defined);
array_name is the name of the array object being declared; and
CAPACITY is the number of values the object can contain.

Figura 3

Sintaxis de la declaración de un arreglo unidimensional en C++[1].

De manera parecida a la declaración de funciones, *type* indica el tipo de datos a almacenar; *array_name* sirve como identificador del arreglo; y *CAPACITY* establece la cantidad de valores que el arreglo puede almacenar. Se debe señalar que, en C++ los arreglos son “zero-based”, es decir, el primer elemento del arreglo es siempre indexado con un cero, de tal manera que si establecemos una capacidad de 6 elementos al arreglo, los índices irían del 0 a 5 de la siguiente manera (ver pp. 457-460) [1].

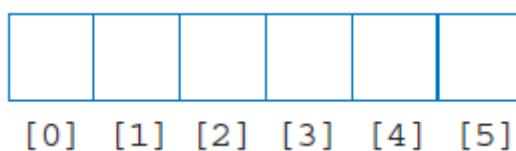


Figura 4

Índices de un arreglo de 6 elementos [1].

Los elementos del arreglo pueden ser inicializados directamente en la declaración del arreglo mediante llaves (“{}”), o pueden ser establecidos posteriormente mediante funciones o directamente accediendo al nombre del arreglo y el índice correspondiente (*array_name[i]*) .



También es posible definir arreglos con más de una “fila” o “columna”, los cuales son conocidos como arreglos multidimensionales. La sintaxis es muy parecida a la de los arreglos unidimensionales, con la única diferencia de que se añaden más dimensiones en la declaración de la siguiente forma.

ARRAY DECLARATION

FORM:

ElementType arrayName[DIM₁] [DIM₂] . . . [DIM_n];

where:

ElementType is any known type;

arrayName is the name of the array being declared; and
each *DIM_i* must be a nonnegative integer (constant) value.

Figura 5

Sintaxis de la declaración de un arreglo multidimensional en C++ (ver pp. 512) [1].

2.4 MEMORIA DINÁMICA CON MATRICES

En primer lugar, se necesita definir los arreglos dinámicos: estos son arreglos cuyas capacidades son especificadas durante el tiempo de ejecución, los cuales pueden ser construidos utilizando las herramientas que C++ proporciona para la reservación y liberación de memoria dinámica: los operadores *new* y *delete*. Por ejemplo, si queremos reservar memoria dinámica para un arreglo de 6 enteros, hacemos las siguientes declaraciones (ver pp. 107- 113)[2]:

```
int * arrayPtr;  
arrayPtr = new int[6];
```

Como se sabe, el operador unario *new* devuelve la dirección basal de este arreglo dinámico de 6 enteros, el cual es almacenado inmediatamente después en el puntero *arrayPtr*.

Ahora bien, se ha decidido comenzar con esta definición ya que, después de todo, una matriz es un arreglo de arreglos, y la asignación de memoria dinámica para este es análoga al de los arreglos unidimensionales como se ve en el siguiente ejemplo.



```

int main()
{
    const int NUM_ARRAYS = 10;
    cout << "How large should the arrays of doubles be? ";
    int capacity;
    cin >> capacity;

    double * arrayPtr[NUM_ARRAYS];
    int i;
    for (i = 0; i < NUM_ARRAYS; i++)
    {
        arrayPtr[i] = new double [capacity];
        cout << "Allocated " << capacity
            << " doubles for i = " << i << endl;
    }
    cout << "All " << NUM_ARRAYS << " arrays of "
        << capacity << " doubles were allocated successfully." << endl;
}
    
```

Figura 6
Código que muestra cómo inicializar una matriz dinámica [2].

Como se ve del ejemplo se declara *arrayPtr* como un arreglo de punteros y después mediante un bucle *for* se asigna memoria dinámica a cada uno de sus elementos o “filas”. Nosotros usaremos una versión alternativa de este método conocido como “método de varios bloques” (revisar Unidad 9), que utiliza, como el nombre indica, varios bloques para reservar la memoria dinámica de las “filas” de la matriz. Primero, se crea la matriz como un arreglo de punteros y se le asigna un tamaño de *fil*sizeof(int*)* bytes en el *heap* y, después, mediante un bucle *for*, se asigna memoria dinámica a cada “fila” de la matriz, como se muestra enseguida.

```

int fil = 2, col = 3;
int** matriz = new int*[fil];
for (int j = 0; j < fil; ++j)
    matriz[j] = new int[col];
    
```

Finalmente, para liberar la memoria dinámica asignada se vuelve a utilizar un bloque *for*, liberando la memoria de cada “fila” de la matriz para después liberar la memoria del arreglo de punteros.

```

for (int j = 0; j < fil; ++j)
    delete[] matriz[j];
delete[] matriz;
    
```



2.5 ARCHIVOS DE TEXTO

Los flujos o *streams* nos proporcionan una manera uniforme de procesar datos desde el teclado o el disco duro hacia el monitor o, de vuelta, al disco duro. Ahora bien, si lo que se quiere es manipular archivos, se deben utilizar los objetos *ofstream* e *ifstream*, que sirven para escribir en, o leer desde archivos, respectivamente; para poder usar estos objetos, debemos asegurarnos de incluir la librería *fstream*, que los incluye (ver pp.624-628)[3].

¿Cómo abro entonces un archivo? Para hacer conexión con un determinado archivo para escritura, primero debemos declarar un objeto *ofstream* y pasárle el nombre del archivo como parámetro de la siguiente manera,

```
ofstream fout("myfile.cpp");
```

El tratamiento es el mismo para la lectura:

```
ifstream fin("myfile.cpp");
```

Nótese que *fout* y *fin* son nombres elegibles por el usuario; y, son estos nombres los que se usarán en la solución del problema, dada su semejanza a los objetos clásicos *cout* y *cin*.

Una vez que se ha asociado a los objetos *ofstream* con el archivo, se pueden usar como cualquier otro objeto *stream*. Finalmente, al terminar de trabajar con los archivos, debemos cerrarlos mediante la función *close()*; esto para asegurarnos que los datos no serán corrompidos y serán enviados al disco duro [3].

Es importante mencionar que al abrir un archivo, el comportamiento por defecto es crearlo si no existe, o destruir el contenido preexistente. En cualquier caso, podemos cambiar este comportamiento mediante la inclusión de un segundo parámetro al declarar nuestro objeto *ofstream*. Dichos parámetros se muestran en la siguiente figura.

- **ios::app**—Appends to the end of existing files rather than truncating them.
- **ios::ate**—Places you at the end of the file, but you can write data anywhere in the file.
- **ios::trunc**—Causes existing files to be truncated; the default.
- **ios::nocreate**—If the file does not exist, the open fails.
- **ios::noreplace**—If the file does already exist, the open fails.

Figura 7

Lista de valores válidos para el segundo parámetro de nuestro constructor de objeto ofstream [3]. Se utilizarán ios::app y ios::trunc en la realización de este proyecto.



3. PLANTEAMIENTO DEL PROBLEMA

“The game of Life, invented by the mathematician John H. Conway, is intended to model life in a society of organisms. Consider a rectangular array of cells, each of which may contain an organism. If the array is assumed to extend indefinitely in both directions, each cell will have eight neighbors, the eight cells surrounding it. Births and deaths occur according to the following rules:

- a. An organism is born in an empty cell that has exactly three neighbors.
- b. An organism will die from isolation if it has fewer than two neighbors.
- c. An organism will die from overcrowding if it has more than three neighbors.

Write a program to play the game of Life and investigate the patterns produced by various initial configurations. Some configurations die off rather quickly; others repeat after a certain number of generations; others change shape and size and may move across the array; and still others may produce “gliders” that detach themselves from the society and sail off into space” (ver pp. 545, Programming in C++) [1].

En 1970, el reconocido matemático inglés John Conway creó un juego al que denominó “vida” (o “life” en inglés). Este juego, que fue dado a conocer a través de la famosa revista de divulgación científica “Scientific American”, pretendía simular el comportamiento de una sociedad de organismos vivientes, su crecimiento, evolución y, eventualmente, extinción [4]. La idea era empezar el juego con un conjunto o “disposición inicial” bastante simple de organismos o “células”, y, a través de un conjunto de reglas cuidadosamente establecidas, ver cómo estas se multiplicaban y desaparecían de manera impredecible. Es interesante remarcar que por aquellos años las computadoras no eran tan accesibles como en nuestros días, por lo que el juego se solía reproducir con ayuda de un lápiz y un papel lo suficientemente grande.



Figura 8
John Conway jugando el juego de la vida [5].



Ahora bien, para entender el juego, se necesitan establecer las reglas, pero primero imaginemos a cada célula como un pequeño cuadrado pintado en una gran hoja o malla cuadriculada, de tal manera que cada célula tiene 8 posibles vecinas: 4 en los lados laterales y 4 en las diagonales (ver Figura 9). Los cuadrados en blanco representan, a su vez, células muertas.

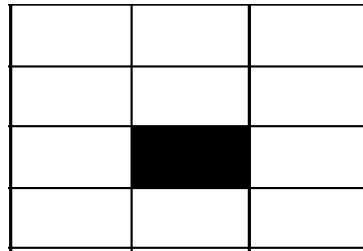


Figura 9
Esquema de representación de una célula.

A primera vista, el esquema de la Figura 7 nos recuerda a un arreglo multidimensional, por lo que parece una decisión razonable elegir uno como nuestra “hoja de juego”. El problema es que esta hoja de juego es, para fines prácticos, infinita, por lo que se intentará representar solo una pequeña parte de esta en el código mediante una matriz cuadrada de considerable tamaño. Finalmente, para las células, se escogerá el carácter asterisco (*). Teniendo ya una representación en mente, ahora sí, se enuncian las reglas como fueron originalmente concebidas [4]:

1. Supervivencia: cada célula con dos o tres vecinas sobrevive en la siguiente generación.
2. Muerte: cada célula con cuatro o más vecinas muere por superpoblación, y cada célula con una sola vecina muere por inanición en la siguiente generación.
3. Nacimientos: cada célula muerta con exactamente 3 vecinas, “revive” en la siguiente generación.

Se debe resaltar que por vecinas nos referimos a las células vivas y por generación a cada “turno” del juego. También es necesario mencionar que el nacimiento, supervivencia y muerte de células se da en simultáneo de tal manera que las células nacientes de una nueva generación no intervienen en la muerte o supervivencia de las células que se tenían inicialmente y viceversa. Para ilustrar mejor el concepto de generación, se jugará con una disposición inicial sencilla (y bastante conocida, por cierto): el “blinker” o “pestañeador”.

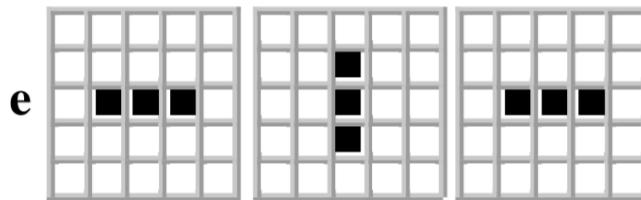


Figura 10

Las dos generaciones de un blinker que se repiten indefinidamente [4].

En la Figura 8, podemos observar en el recuadro de la izquierda, 3 células vivas ordenadas contiguamente de manera horizontal, esta es nuestra primera generación. Lo que se hace a continuación es identificar a todas las células que morirán, que en este caso son las dos en los extremos (ambas tienen solo un vecino), luego la que sobrevivirá, la del centro (tiene dos vecinas), y finalmente a las que nacerán, una arriba y una abajo de la célula central (tienen como vecinas a todo el arreglo inicial). Una vez hecha esta diferenciación, se procede a ejecutar a las células que deben morir, mantener a las supervivientes y establecer a las nacientes, como se ve en el recuadro del centro: esta es nuestra segunda generación. Como se puede observar, la figura ha rotado 180 grados, por lo que es de esperar que la tercera generación también lo haga. A este tipo de disposiciones que repiten su comportamiento cada cierto número de generaciones se les conoce como periódicas [4].

En resumen, como se ha visto con este ejemplo, lo que en realidad se hace al cambiar de generación es aplicarle, por separado, las reglas establecidas al mismo arreglo inicial, solo modificándolo al final de la generación. Esta concepción del problema arroja luces a cómo abordar satisfactoriamente la elaboración del código: habíamos elegido una matriz como nuestro tablero de juego y el carácter * como representación de la célula, las reglas ya han sido presentadas de tal manera que solo faltaría aplicarlas a la matriz de juego dada una disposición inicial. Estas reglas, como acabamos de explicar, siempre se aplican al mismo arreglo de la generación presente, por lo que resulta bastante lógico hacer lo mismo con las iteraciones de matrices que representen generaciones.

Finalmente, queda el tratamiento de esa “disposición inicial”: ¿Cómo se representará fidedigna y eficientemente la jugada libre del usuario? Después de todo, el juego de Life es un juego de cero jugadores, es decir, un juego que solo necesita un input inicial del usuario, prescindiendo de cualquier entrada de datos posterior. Para la resolución de este aspecto del problema, se debe tener en cuenta que las entradas del usuario son libres y están limitadas tan solo por la imaginación del jugador. Se decide entonces trabajar con un conjunto de datos que, de alguna manera, representen estas jugadas creativas y puedan serle “comunicadas” a nuestro programa para que éste los procese adecuadamente de acuerdo a las reglas señaladas anteriormente. Siendo que se han elegido caracteres * para representar las células, nuestros conjuntos de datos serán archivos de texto “input”, cuidadosamente ordenados, de caracteres y espacios.

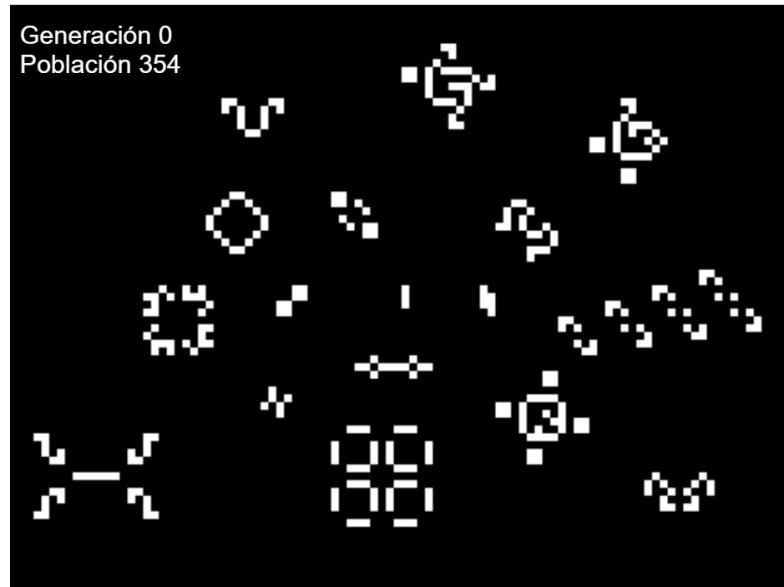


Figura 11

Las jugadas iniciales no están limitadas más que por la imaginación del usuario [6].

4. SOLUCIÓN DEL PROBLEMA

El método elegido para simular el juego de la vida en C++ tiene muy en cuenta las reglas que rigen el comportamiento de las células y cómo estas se aplican dada una determinada disposición inicial. Como se estableció anteriormente, se escogió una matriz como tablero de juego y el carácter "*" para la representación de células. La idea central detrás del método elegido es revisar e identificar las células sobrevivientes/nacientes y/o las que morirán en la siguiente generación, es decir identificar todos los cambios en la disposición de células actual, y con esa información, aplicar simultáneamente los criterios de las reglas del juego de la vida al arreglo de células actual, que se convertirá a su vez en la disposición inicial de la siguiente generación. Para tal fin se utiliza principalmente la técnica de recursividad y la aplicación de múltiples condicionales. Por otro lado, para la lectura de las jugadas de los usuarios se han preparado previamente cierto número de archivos de texto que representan diversas jugadas conocidas (como la “nave espacial”), las cuales son leídas directamente por nuestro programa y procesadas en un arreglo dinámico multidimensional, evitando así la tarea de tener que inicializar “manualmente” alguna jugada particular.

Se adjuntan, junto con el informe, el código en c++ y los archivos de texto complementarios para la exitosa prueba del programa. Sin más, se procede, primero a presentar el pseudocódigo, y luego a explicar brevemente los aspectos más relevantes del programa, así como el por qué de su implementación. Se recomienda al lector tener a la vista el código para que pueda ubicarse correctamente cuando se mencione el número de línea.

FACULTAD DE CIENCIAS

Escuela Profesional de Ciencia de la Computación



Algoritmo THE GAME OF LIFE

```

Para int i=0 Hasta CAPACIDAD Con Paso 1
FinPara
inicializarlife
Para int i=0 Hasta CAPACIDAD Con Paso 1
FinPara
inicializarcopia
cout<<"Ingresar jugada: "
cin>>jugada
ifstream finjugada
assertfin.is_open()
Para int i=0 Hasta CAPACIDAD Con Paso 1
    Para int j=0 Hasta CAPACIDAD Con Paso 1
        Si ((ch=fin.get())!=10)
            life[i][j]=ch
        FinSi
    FinPara
FinPara
fin.close
ofstream fout"respaldo",ios::trunc
    assertfout.is_open()
fout.close
mostrarlife
cout<<endl
generacionlife,copia
Para int i=0 Hasta CAPACIDAD Con Paso 1
    delete[] copia[i]
FinPara
delete[] copia
Para int i=0 Hasta CAPACIDAD Con Paso 1
    delete[] life[i]
FinPara
delete[] life

FinAlgoritmo
SubProceso INICIALIZAR(** P)
    Para int i=0 Hasta CAPACIDAD Con Paso 1
        Para int j=0 Hasta CAPACIDAD Con Paso 1
            p[i][j]=' '
        FinPara
    FinPara
FinSubProceso
SubProceso MOSTRAR(** P)
ofstream fout"respaldo",ios::app
    assertfout.is_open()
fout<<"Generacion "<<Contador<<":\n"=fout<<"Generacion "<<Contador<<":\n"+1
Para int i=0 Hasta CAPACIDAD Con Paso 1
    Para int j=0 Hasta CAPACIDAD Con Paso 1
        cout<<p[i][j]<<" "
        fout<<p[i][j]<<" "
    FinPara
    cout<<endl
    fout<<endl
FinPara
fout.close
  
```

FACULTAD DE CIENCIAS
Escuela Profesional de Ciencia de la Computación



```

FinSubProceso
SubProceso GENERACION(* P,* Q)
    cout<<"Mostrar variableReservada siguiente generacion?Introduzca cualquier caracter para continuar o
escriba no"<<endl
        cin>>respuesta
        Si ((respuesta[0]==n | respuesta[0]==N) y respuesta[1]==o)
            FinSi
            Para int i=0 Hasta CAPACIDAD Con Paso 1
                Para int j=0 Hasta CAPACIDAD Con Paso 1
                    q[i][j]=p[i][j]
                FinPara
            FinPara
            Para int i=1 Hasta CAPACIDAD-1 Con Paso 1
                Para int j=1 Hasta CAPACIDAD-1 Con Paso 1
                    Si (q[i][j]==*)
                        Si (vecinos(qij)<2 | vecinos(qij)>3)
                            //muertes
                            p[i][j]=' '
                        FinSi
                    SiNo
                        Si (vecinos(qij)==3)
                            //nacimientos
                            p[i][j]='*'
                        FinSi
                    FinSi
                FinSi
            FinPara
            mostrarP
            cout<<endl
            generacionP,q
        FinSubProceso
        SubProceso VECINOS(** P, FIL, COL)
            Si (p[fil][col-1]==*) cont++
                Si (p[fil][col+1]==*) cont++
                    Si (p[fil-1][col]==*) cont++
                        Si (p[fil+1][col]==*) cont++
                            Si (p[fil+1][col+1]==*) cont++
                                Si (p[fil-1][col-1]==*) cont++
                                    Si (p[fil-1][col+1]==*) cont++
                                    Si (p[fil+1][col-1]==*) cont++
                                    FinSi
                                FinSi
                            FinSi
                        FinSi
                    FinSi
                FinSi
            FinSi
        FinSubProceso
    FinSi
}

```

Listado 1: Pseudocódigo del algoritmo que soluciona el problema



En primer lugar, se declaran las librerías a utilizar: `iostream`, `fstream` (para el uso de archivos de texto) y `cassert` (para asegurarse de la correcta apertura de estos archivos). En la línea 5 declaramos la constante `CAPACIDAD` con un valor de 40 y, aunque este valor se puede ajustar si se desea tener un “tablero” más grande, se debe tener en cuenta que también deben modificarse los archivos de texto “input”, como se explicará más adelante.

A continuación, se declara un contador inicializado en cero, que servirá para nuestro archivo “output” llamado `respaldo`, y los prototipos de las funciones `mostrar`, que imprime los elementos de la matriz tanto en el monitor como en el archivo `respaldo`; `generacion`, que servirá para “jugar”; `vecinos` que servirá para contar el número de vecinos de una célula; e `inicializar`, que “llena” de caracteres espacio a nuestro tablero de juego. Todas estas funciones tienen como parámetros al menos un arreglo de punteros a carácter, dada la naturaleza del tablero de juego, como se verá a continuación. En la línea 13, se declara `life` como una matriz bidimensional de caracteres y se le asigna espacio de memoria dinámica utilizando el método de varios bloques (revisar fundamento teórico). ¿Por qué este método? Dada la naturaleza del tablero de juego, o se tienen células vivas, representadas por `*`, o se tienen células muertas, representadas por el carácter espacio, de tal manera que se está haciendo uso de la totalidad de la matriz `life`. Inmediatamente después, en la línea 17, se llama a `inicializar`, para no perder la buena costumbre de inicializar un puntero en cuanto se declare. Repetimos estos dos últimos procedimientos con una matriz `copia`, que será de utilidad más tarde. Luego, en la línea 25 se declara un literal de cadena, `jugada`, el cual debe ser ingresado por el jugador. Este literal de cadena, necesariamente, debe ser el nombre de un archivo de texto en el que esté “escrita” la jugada inicial representada como una matriz de caracteres con unas dimensiones particulares, en este caso de `CAPACIDAD` filas y columnas.

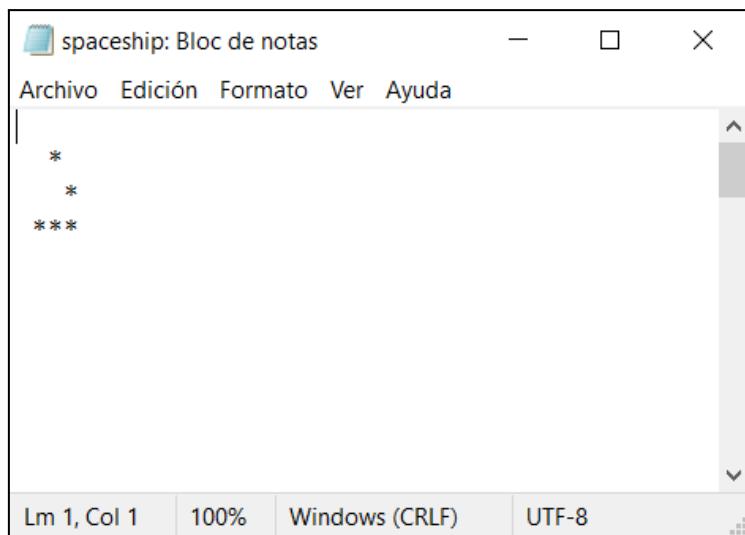


Figura 12

Archivo de texto “`spaceship.txt`” en el bloc de notas. Se ha llenado de caracteres espacio todas las “células” muertas.



Es por esta razón que se resaltaba al inicio la importancia de modificar los archivos de texto “input” si se deseaba aumentar o disminuir el valor de `CAPACIDAD`: dado que en las líneas 29 y 30 se hace conexión con el archivo para su lectura mediante el objeto `fin`, y después, mediante un bucle, se leen directamente del archivo los elementos de `life` (ignorando los saltos de línea, carácter 10 en código ASCII), se debe asegurar que estos “encajen” con los elementos del archivo; de otra manera, la lectura y posterior asignación estarían desordenadas y se desfiguraría la jugada inicial del usuario. A continuación, en las líneas 43 y 44, se hace conexión con el archivo `respaldo` para escritura mediante el objeto `fout`, haciendo uso del parámetro adicional `ios::trunc`, que elimina los contenidos existentes del archivo (este comportamiento es el predeterminado, pero se le incluye por fines didácticos). ¿Por qué se necesita hacer esto? `respaldo` es nuestro archivo “output”, que guarda el historial generacional de la jugada anterior, cualquiera que haya sido esta y, dado que en la línea 46 se llama a la función `mostrar`, que escribe tanto en el monitor como en `respaldo` la evolución generacional de la jugada, se necesita usar del parámetro `ios::app` cada vez que se llame a esta función, apilando las generaciones una tras de otra en `respaldo` (he aquí la funcionalidad del contador mencionado al inicio, que escribe en el archivo la edad de cada generación). Para finalizar, se llama a la función `generacion`, pasándole `life` y `copia`, y se devuelve la memoria dinámica de estas matrices bidimensionales de caracteres según el método de varios bloques.

Ahora bien, lo que hace `generacion` es, efectivamente, producir nuevas generaciones. Se comienza en la línea 85 preguntando al jugador si desea ver la siguiente generación, para lo cual se define una cadena de caracteres `respuesta` que guardará su respuesta. Si el jugador responde negativamente se terminará la ejecución de la función mediante `return`. Debido a la naturaleza del juego, las generaciones se seguirán mostrando indefinidamente, por lo que se establece este condicional como una manera de terminar el programa. Como se sabe, cuando algún `return` se encuentra en la función, ésta termina de ejecutarse inmediatamente y todas las sentencias que estaban después, son ignoradas.

A continuación, en las líneas 91 a 95, se le asigna a la matriz `q` los elementos que `p` tiene en esa instancia. Como se recuerda, del planteamiento del problema, lo que en realidad se hace es aplicar las reglas por separado al mismo arreglo inicial, razón por la cual creamos esta copia `q` de la actual generación. Después, en las líneas 96 a 109, evaluamos, mediante `vecinos`, el número de células vivientes adyacentes a cada célula viviente de `q`, de tal manera que, si el número de vecinos es menor a 2 o mayor a 3 (línea 99), se modifica la matriz original que fue recibida como parámetro, “ejecutando” a las células que ocupaban sus lugares (se les define como el carácter espacio). Análogamente, para las células nacientes, si el número de vecinos es exactamente igual a 3, se reemplazan los espacios vacíos de la matriz parámetro por células vivientes con el carácter *. Finalmente, se vuelve a llamar a `mostrar`, ingresando la misma matriz original, ahora modificada, y se emplea la recursividad reutilizando la función `generacion`, repitiendo el proceso para la siguiente generación.



5. RESULTADOS

- En primer lugar, se logra resolver satisfactoriamente el problema planteado: el algoritmo propuesto en la sección anterior permite mostrar la evolución generacional de cualquier población inicial particular hasta donde el usuario decida. Se debe volver a recalcar que, a pesar del correcto funcionamiento del programa, la memoria que se ha reservado es limitada, ergo, las dimensiones de nuestro tablero también lo son, por lo que se apela a la discreción del jugador a la hora de asegurarse de no hacer crecer a la colonia de células más de lo que CAPACIDAD lo permite. En cualquier caso, el valor de esta constante se puede modificar, un beneficio que se ha obtenido del uso de la memoria dinámica. A continuación, se muestra el funcionamiento del programa para una jugada específica, el famoso “spaceship”.

The screenshot shows two instances of the CLion IDE interface. Both windows have 'Run' tabs with 'untitled' selected. The top window's run output shows the command 'Ingresar jugada: spaceship.txt' followed by the initial state of the pattern: three asterisks on each of the first three lines. The bottom window's run output asks 'Mostrar siguiente generacion?' and 'Introduzca cualquier caracter para continuar o escriba no)' followed by the user input 'si'. Below this, it shows the next generation of the pattern, which is a spaceship ship moving to the right, consisting of three rows of three asterisks each.

Figura 13

Se ingresa el nombre del archivo “spaceship.txt” desde el teclado y se observa la evolución de la jugada hasta que el usuario decida lo contrario.



- En segundo lugar, el jugador puede proponer cualquier arreglo de células, siempre y cuando no supere las dimensiones de nuestro “tablero” (como ya se recalcó), y siga cuidadosamente un orden a la hora de ingresar los caracteres en un archivo de texto que hemos llamado “jugada_libre”: por cada carácter * que se escriba se debe, inequívocamente, quitar un carácter espacio; esto para mantener el orden señalado a la hora de que el programa “lea” nuestra jugada (revisar sección 4).

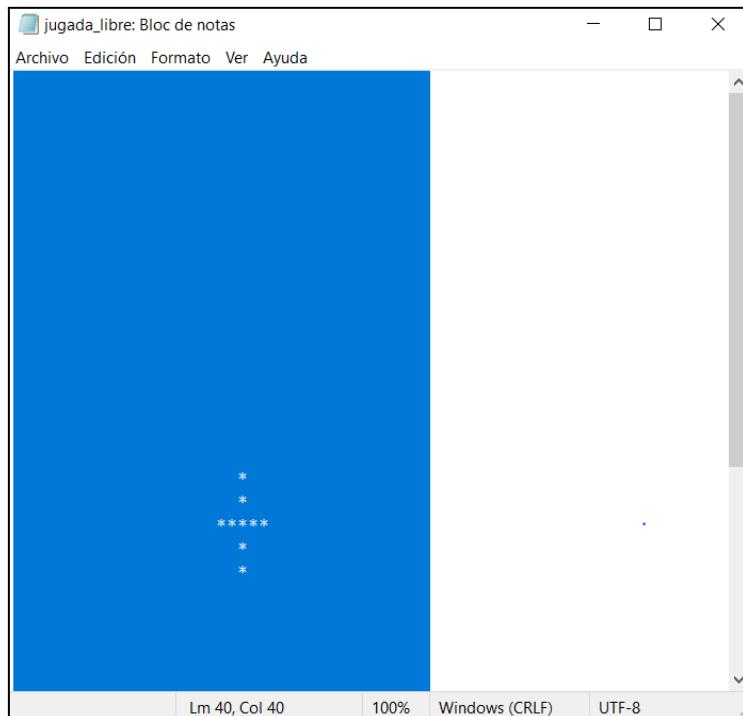


Figura 14

El jugador puede crear colonias y evaluar su comportamiento desde el archivo jugada_libre.txt, asegurándose de conservar las dimensiones de CAPACIDAD.

- Finalmente, si el jugador así lo desea, se puede guardar el registro de la evolución poblacional de una determinada colonia de células, gracias al archivo creado *respaldo*, que guarda el historial generacional de la última jugada. Como se mencionó en el planteamiento del problema (ver sección 3), existen una infinidad de formas que pueden crearse, algunas de las cuales podrían serle de utilidad al jugador para su revisión.

FACULTAD DE CIENCIAS
Escuela Profesional de Ciencia de la Computación



<p>respaldo: Bloc de notas</p> <p>Archivo Edición Formato Ver Ayuda</p> <p>Generacion 0:</p> <p style="text-align: center;">* * * * * *</p>	<p>respaldo: Bloc de notas</p> <p>Archivo Edición Formato Ver Ayuda</p> <p>Generacion 1:</p> <p style="text-align: center;">* * * * * *</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------

Figura 15
El archivo respaldo permite al jugador guardar el historial de su colonia de células.



6. BIBLIOGRAFÍA

A continuación se muestran las referencias del fundamento teórico recogidas de 3 diferentes libros. El resto de referencias son suplementarias y remiten a fotografías o artículos científicos.

- [1] Nyhoff, L. (2012). Programming in C++ for Engineering and Science. CRC Press.
- [2] Nyhoff, L. (2005). ADTs, Data Structures, and Problem Solving with C++. Prentice Hall, 2nd edition.
- [3] Liberty, J. and Jones, B. (2004). Sams Teach Yourself C++ in 21 Days. Sams, 5th edition.
- [4] Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4), 120-123.
- [5] John Horton Conway: The world's most charismatic mathematician | Siobhan Roberts. (2015, July 23). Retrieved from <https://www.theguardian.com/science/2015/jul/23/john-horton-conway-the-most-charismatic-mathematician-in-the-world>
- [6] Juego de la vida. (2021, January 21). Retrieved from https://es.wikipedia.org/wiki/Juego_de_la_vida