UNIDAD 8

Creación Dinámica de Objetos.

1.- **Introducción.**

Los seres humanos se comunican entre si, principalmente a través del lenguaje, Un lenguaje no solo nos da herramientas para interactuar, sino que también define las interacciones y maneras de pensar. Si se entienden las limitaciones del lenguaje, es posible entender algunas de las limitaciones en la forma de pensar y la creatividad.

Los lenguajes de computadora han incorporado ciertas tendencias encaminadas a una determinada forma de pensar, diseñar y programar, lo que en cierta forma limita la creatividad y forma de pensar de los programadores, en términos del camino a seguir. Como resultado hay cierta clase de problemas que son imposibles de resolver.

las tendencias serán analizadas aquí, pueden ser el punto de partida, como la idea de que siempre se conocen todos los factores cuando se escribe un programa. Esta forma de hacer las cosas se extrapola al diseño de programas (siempre se conocen todos los problemas que el programa deberá resolvar cuando se diseña). En esta Unidad nos vamos a enfocar a la manera en que las variables son tratadas en otros lenguajes, como afectan en la programación y las inovaciones introducidas en C++, que nos abren el panorama y la forma de enfocar ciertos tipos de problemas.

2.- **Tendencias de algunos de los lenguajes más populares.**

Muchos de los lenguajes más populares, como FORTRAN, Pascal y C, parten de la idea de que siempre se conoce cuantas variables van a ser necesarias cuando el programa esta corriendo y cuando ellas van a ser creadas y destruidas, Cuando se encuentra un problema el cual depende de otros factores para la determinación de cuantas variables son necesarias y cuando ellas deberán ser creadas y destruidas, no es posible resolver el problema, y si se logra hacerlo se hace de una manera complicada y usando un diseño casi inflexible.

Por ejemplo, considere el juego **life** (juego de la vida). Es un simulador que modela los diseños de población de una 'criatura', la cual vive en una retícula bidimensional. Las reglas para la existencia de esa criatura son muy simples. Si hay un cierto número de criaturas unidas a un punto vacio de esa retícula, nace una nueva criatura. Si una criatura tiene demasiados 'vecinos' o muy pocos, morirá. Mediante el cambio de las condiciones iniciales y las reglas, se pueden generar otras clases de diseño facinantes.

En la simulación **life**, los objetos (criaturas) estan apareciendo y desapareciendo constantemente. La cantidad y el tiempo de vida de los objetos no se establece por un

diseño fijo del programa, sino que depende del número de objetos que existan en torno a un objeto dado. Este es un ejemplo de un problema de simulación, sin embargo, la mayoría de estos problemas son mucho más complicados que **life**. Aunque **life** ha sido programado en muchos lenguajes, un problema de simulación '**real**' es mucho más manejable cuando se escribe en un lenguaje orientado a objetos.

Es evidente que la limitación arbitraria, de tener que conocer la cantidad, el punto de creación y el tiempo de vida de un objeto evita un gran número de problemas. La forma en que uno ha aprendido a pensar, como consecuencia de un lenguaje de programación particular, puede evitar el concebir algunas soluciones interesantes.

C y Pascal tienen un método para crear espacio para las variables en tiempo de ejecución. (FORTRAN, no lo tiene). Esta característica de asignación dinámica de memoria, permite al programador escribir directamente en memoria y luego usar dicha área como una variable. Como se puede ver, el soporte para crear variables en tiempo de ejecuación es una buena alternativa; el crear un objeto en tiempo de ejecuación es tan fácil como hacerlo en tiempo de compilación.

Lenguajes como LISP y Smalltalk, exageran, creando todas sus variables en tiempo de ejecución. El tiempo de vida de estas variables no se determina por reglas de ámbito normales, este tipo de variables no se destruyen en función de su ámbito, pueden crecer hasta el punto de llenar la memoria. Cuando la memoria se encuentra saturada, estos lenguajes ejecutan una rutina de recogida de información no válida, de tal manera de dejar libre el espacio ocupado por aquellas variables no utilizadas.

Los lenguajes con procedimientos de recoger información no válida, liberán al programador de aspectos relacionados con el tiempo de vida de una variable, sin embargo, crean otros problemas.

3.- **Creación dinámica de objetos.**

C++ pertenece a un grupo intermedio, entre aquellos lenguajes que soportan asignación dinámica de memoria, en los cuales todas las variables son asignadas dinamicamente, Nosotros llamaremos al enfoque de C++, **creación dinámica de objetos**.

C++ soporta objetos

* Con tiempo de vida limitado.
  (son objetos asignados a la pila),
* Con un tiempo de vida arbitrario.
  (objetos asignados al heap).

C++ es diferente que Pascal y C ya que no solo asigna memoria a un objeto, también lo inicializa.

Cuando se usa creación dinámica de objetos en C++, además de asignar memoria se inicializa el objeto.

Cuando se asigna memoria dinámica a un tipo predefinido, no se realiza una inicialización, por ello, se puede seguir usando el término **'Asignación dinámica de memoria'** para un tipo predefinido.

Debido a que C++ soporta tanto objetos con tiempo de vida limitado, como objetos con tiempo de vida arbitrario, el programador esta obligado a conocer un poco más acerca de como opera un programa, de tal manera, de que en un momento dado podamos seleccionar entre, objetos basados en el stack y objetos basados en el heap. Esta decisión de diseño del lenguaje es considerada a menudo como "repugnante", para los usuarios de los lenguajes más puros orientados a objetos, (aquellos que contienen recogedores de información no válida). Como se puede ver, sin embargo, le brinda al programador la posibilidad de la optimización del programa, en velocidad, en aplicaciones de tiempo real.

Para entender la creación de objetos dinámicos, es necesario contar con algunas bases.

3.a) La pila (STACK).

Cuando se define un objeto en tiempo de compilación, se escribe:

```
void uno() {
     int obj = 100;
...
...
}
```

Entonces el compilador genera código para mover el apuntador de la pila hacia abajo, de tal manera de crear espacio sobre ésta para almacenar el valor entero y deposita el valor de 100, en ese espacio. Mientras la variable **obj** es utilizada, se genera código para localizar su posición. Note que el espacio de almacenamiento es asignado a partir de la declaración de las variables, hasta que se sale del ámbito.

Esta acción tiene lugar cada vez que se llama la función uno(), Entonces, la memoria se crea siempre y cuando la función es llamada, sin tener en cuenta la posición del apuntador de la pila en el momento de la llamada. Funciones en C y en C++ que solo usan variables locales no dependen del estado del programa cuando la función es llamada, por lo tanto, pueden ser llamadas en cualquier punto del programa. Se dice que este tipo de funciones (que solo usan variables locales) son re-entrantes.

C y C++ también pueden manejar funciones recursivas, como se ve en el siguiente ejemplo:

```
// Programa 62, funciones recursivas.
#include <iostream.h>

const xa = 10;

void frec(int pun)
{
    if(pun <= xa)
        frec(++pun);
    cout << "Valor de pun: " << pun << endl;
}

void main()
{
    frec(0);
}
```

OBSERVACIONES.

La salida del programa nos resulta:

```
Valor de pun: 11
Valor de pun: 11
Valor de pun: 10
Valor de pun: 9
Valor de pun: 8
Valor de pun: 7
Valor de pun: 6
Valor de pun: 5
Valor de pun: 4
Valor de pun: 3
Valor de pun: 2
Valor de pun: 1
```

Note que el valor de **pun** es impreso en sentido contrario de como fué incrementado, lo que significa que la función **frec()** fué solo llamada en la primera etapa, sin que se imprimiera nada, cuando la función recursiva llega a la base, se inicia la segunda etapa, llamando a impresión, cada vez que se sale de una llamada de la función **frec()**, hasta que se regresa a **main()**.

Cuando un ámbito termina (con **}**), el compilador genera código para mover hacia arriba el apuntador de pila, con lo que se elimina automáticamente cada una de las variables definidas dentro del ámbito. Cuando una variable local, sale de ámbito, con

lo cual la memoria asignada para dicha variable, se libera automáticamente. En C++ se llama al destructor (si existe), antes que sea movido el apuntador de la pila, C++, no solo libera el espacio ocupado por la variable, también limpia la variable.

3.b) El tipo de almacenamiento libre.

El puntero de pila se mueve hacia abajo, tanto cuando se almacena en la pila la dirección de retorno, como cuando se aparta memoria para almacenar las variables locales, por lo tanto, cuando se llama una función, el puntero de pila siempre avanza hacia abajo.

En la otra mano, tenemos un espacio de memoria llamado el **heap**. Cuando se requiere memoria se le solicita al **heap**, cuando ya no se necesita dicha memoria es necesario liberarla.

En las librerias de ANSI C, se tienen las funciones **malloc()**, **calloc()** y **realloc()** para pedir memoria al heap y **free()** para liberarla.

En C++ el concepto de asignación dinámica de memoria, se considera como el corazón del lenguaje. C++ no solamente ha definido las palabras reservados **new** y **delete** para solicitar memoria y para liberar memoria, al crear un objeto con **new**, también llama al constructor, y al destruir el objeto con **delete**, también es llamado el destructor. Esto es casi comparable a la creación de una variable local con **new**, y el llamado a **delete**, es casi similar a la salida de ámbito de una variable local. Por esta razón, el término "Creación de objetos dinámicos" se escogió en vez del término "Acceso dinámico a memoria".

Cuando se llama a **new**, regresa un puntero al primer byte del objeto y **delete**, libera memoria a partir de la dirección regresada por **new**, note que aquí se esta manejando una dirección, en vez de un objeto completo, los objetos creados en el **heap** pueden ser manejados como si fueran variables locales.

A continuación se presenta un ejemplo, en donde se compara un objeto local con uno creado en el **heap**.

```
// Programa 63, manejo de la pila y el heap.
#ifndef PILAH_H_
#define PILAH_H_
#include <iostream.h>

class pilaheap {
    int i;
public:
    pilaheap(int x = 0) {
        cout << "Constructor llamado con argumento " << x
             << endl;
```

```
        i = x;
    }
    ~pilaheap() {
        cout << "Destructor llamado por objeto local: i = "
             << i << endl;
    }
    void imprime() {
        cout << "pilaheap::imprime(): i = " << i << endl;
    }
};

#endif // PILAH_H_
```

A partir de la definición de la clase, se pueden crear objetos tanto en la pila (variables locales), como en el heap.

```
// Programa 63, uso de la pila y el heap.
#include "plcp63.h"

void vel()
{
    pilaheap local(10);  // Se crea un objeto local.
    pilaheap *free = new pilaheap(20); // objeto en el heap
    local.imprime();
    free -> imprime();
}

void main()
{
    vel();
}

// Note que el destructor es llamado solamente una vez para
// el objeto local. El apuntador "free" sale del ámbito, pero
// el objeto al que apunta permanece válido
```

OBSERVACIONES.

La salida nos resulta:

```
Constructor llamado con argumento 10
Constructor llamado con argumento 20
pilaheap::imprime(): i = 10
pilaheap::imprime(): i = 20
```

```
Destructor llamado por objeto local: i = 10
```

En la salida podemos observar que solo se llamó al destructor en el caso de la variable local.

Cuando se crea un objeto usando **new**, se dá el nombre de la clase, seguida por una lista de argumentos conteniendo los argumentos del constructor (si no hay argumentos, la lista estará vacia). El almacenamiento es asignado al heap y entonces el constructor es llamado para depositar los argumentos de la lista.

Es muy importante notar que el destructor es llamado automáticamente y solamente por el objeto basado en la pila, cuando sale fuera de ámbito. En el caso de los objetos creados en el heap, el programador es el responsable de destruirlos, si no son destruidos, el apuntador sigue existiendo fuera del ámbito, como sucedió en el programa anterior. Es lo peor que puede suceder, ya que el apuntador salió fuera de ámbito, y ya no se tiene acceso a él, y el objeto aun sigue existiendo y no puede ser destruido. Es importante destruir el objeto antes de que se pierda el enlace con él.

Veamos la siguiente aplicación de la clase pilaheap.

```
// Programa 63A, uso de la clase pilaheap.
// Se usa un destructor para el objeto almacenado en el heap
#include "plcp63.h"

void vet()
{
    pilaheap *free = new pilaheap(20);
    free -> imprime();
    delete free; // Al destruir el objeto se deberá usar la
}                // dirección de inicio del objeto.

void main()
{
    vet();
}
```

OBSERVACIONES.

La única modificación realizada al programa PLCP61.CPP fué la liberación de memoria del objeto almacenado en el heap, mediante la sentencia delete, lo cual fué realizado antes de que el puntero saliera de su ámbito.

3.c) Life: Un framework para simulación y modelado.

Muchos sistemas que simulan o modelan el mundo real, requieren que las unidades de información que representan el proceso puedan ser creadas y destruidas en tiempo de ejecución. La creación y destrucción de estas unidades es gobernada por un criterio relacionado directamente con el proceso, más que los ámbitos en el código fuente. La creación de objetos dinámicos en C++ es ideal para resolver este tipo de problemas.

En esta sección se presenta una versión del famoso programa **life**, pero con un nuevo giro, ya que es diseñado e implementado usando el paradigma de programación orientada a objetos. Es muy fácil cambiar las reglas para la simulación. Por ejemplo, se puede querer dotar a cada elemento (llamado un lifeUnit) en el programa con una cierta duración de vida, de tal forma que si una **LifeUnit** es demasiado joven o demasiado grande no puede tener descendencia. Se podría permitir que una **LifeUnit** fuera móvil o que varias **LifeUnit** se unieran para formar tribus. Debido a que el sistema principal simplemente pregunta a una **LifeUnit** si vivirá para la siguiente generación, la "regla" para Life se enfoca en un lugar, y puede ser cambiada facilmente. Se pueden añadir nuevas clases de reglas; si se mueve una **LifeUnit**, se cogerá una plaga bubónica, generará un millón de nuevos pesos, se casará, comprará un perro, etc. Las reglas pueden ser probadas varias veces durante el transcurso de vida ("**LifeUnit** suspende la licencia de conducir"), o por varias otras razones ("**LifeUnit** sobrealimenta a los peces dorados. **LifeUnit** tira a los peces dorados y compra hamster".) Debido a la naturaleza del diseño orientado a objetos, no se esta limitado cuando se desea extender la simulación.

No hay nada particularmente secreto o inspirado acerca de la manera en que este programa fué diseñado. Es una aproximación que a simple vista trabaja, pero puede tener algún tipo de limitaciones para un sistema en particular que se quiera modelar. Por ejemplo, si la idea del programador con respecto a una **LifeUnit** es mucho más amplia, y puede desear modelar a otros animales y hasta al hombre, y limitar sus actividades al tipo de animal, ("Un perro no puede comprar el periódico"). Para este tipo de problemas se deberá usar herencia y funciones virtuales, de tal manera que tanto el humano como el perro, heredaran desde **LifeUnit** y tendrán algunas cosas en común y algunas otras serán diferentes. Siempre que se desee cambiar el diseño para ajustarlo a algún problema determinado, será necesario estudiarlo previamente.

El programa para su representación en pantalla usa funciones globales de manejo de pantalla, como son; clrscr() y gotoxy().

3.d) Una clase para representar una entidad de vida única.

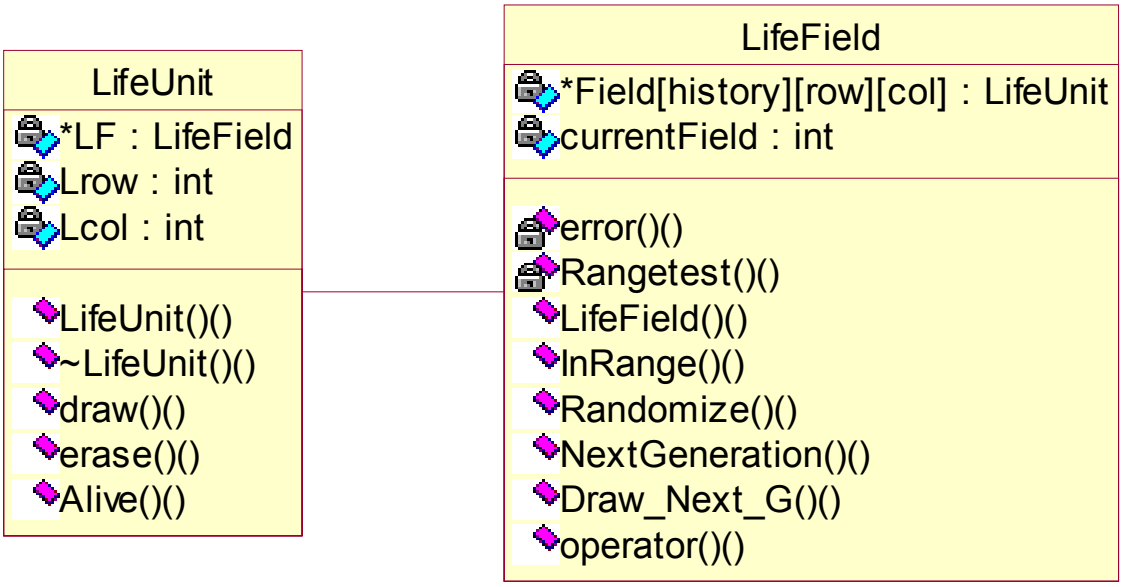
Hay 2 clases en el programa **life**. La clase **LifeUnit**, la cual representa una entidad individual, y la **LifeField**, el cual es un arreglo de 2 dimensiones donde viven

todos los **LifeUnit**. Cada **LifeUnit** sabe en que **LifeField** está y su posición en el campo, por lo que si se pregunta a una **LifeUnit** si va a vivir en la siguiente generación, ella verá a su alrededor cuantos vecinos hay cerca de ella. Una **LifeUnit** puede dibujarse y borrarse a si misma y con **alive()** se verá si vivirá en la siguiente generación.

El archivo de cabecera para la clase **LifeUnit** mostrado aquí, también declara una función global **Fertile()**, la cual es usada con la dirección de una celda vacia, para ver si una nueva **LifeUnit** va a aparecer en la siguiente generación, ésto es, si va a haber un nacimiento.

constantes utilizadas.	Reglas
Minimum Neighbords = 2	Sobrevive
Maximum Neighbords = 3	
Minimum Parents = 3	Nace
Maximum Parents = 3	

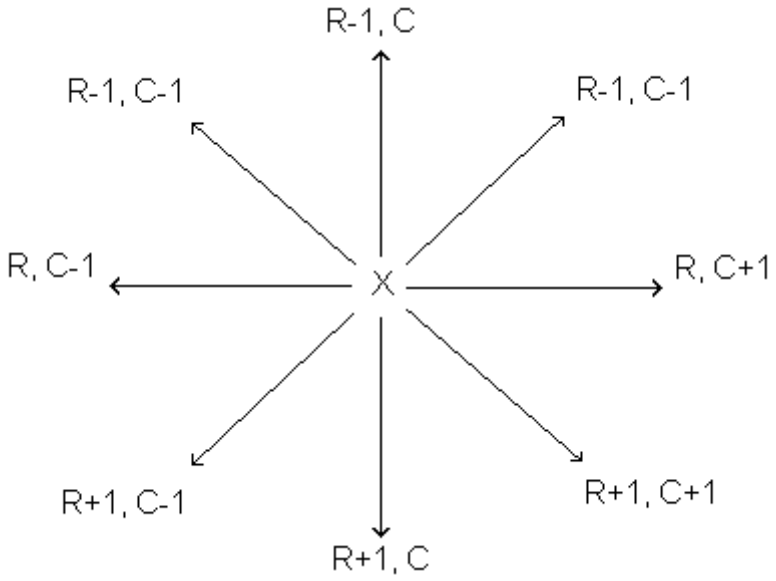
Estructura del programa



funciones globales:

```
fertile();
NeighborAlive();
SurroundingUnit();
```

Operación de la función SurroundogUnit();



Resumen de datos miembro y funciones miembro.

LF	Dirección del objeto objeto de la simulación.
LRew, LCol	Posición de la LifeUnit en la retícula.
LifeUnit()	Almacena las coordenadas de la LifeUnit y la dibuja.
~LifeUnit()	Coloca un CERO donde había una X.
Alive()	Analiza si una LifeUnit pasará a la siguiente generación.
LifeField()	Inicializa el arreglo con CERO.
error()	Envia un mensaje de error.
operator()	Dandole las coordenadas regresa: - LifeUnit - 0
InRange()	Verifica que se esté dentro del rango del arreglo. Regresa un ESTADO.
Rangetest()	Verifica que se esté dentro del arreglo, llama a la función error().
Randomize()	Lena de manera aleatoria el campo con LifeUnit's
NextGeneration()	Analiza cada punto del arreglo: - Si esta vacio, ve si se reunen las condiciones para un nacimiento. - Si existe una LifeUnit analiza si pasará a la siguiente generación.
Draw_Next_G()	Dibuja la siguiente generación en pantalla.

Funciones globales.

Fertile()	Analiza si existen las condiciones propicias para un nuevo nacimiento.
NeighborAlive()	Ve si una celda se encuentra ocupada.
SurroundingUnit()	Cuenta el número de de LifeUnit que hay alrededor de un punto determinado del campo.

```
// Archivo de cabecera de Life
// Programa PLC64A.H
#ifndef LIFEUNIT_H_
#define LIFEUNIT_H_
#include <stdio.h>
#include <conio.h>
#include <iostream.h>

// Caracter para indicar que una LifeUnit esta viva.

const char LifeChar = 'X';
const char birthChar = 'S';
const char DeathChar = '0';

class LifeField;

class LifeUnit {
     // Una LifeUnit sabe en que cuadrante esta y su
localización.
     LifeField *LF;
     int LRow, LCol; // Localización sobre la pantalla.
public:
     void draw(char ch) {
     gotoxy(LCol, LRow);
     printf("%c", ch);
     }
     void erase() {
     // Pone un '0' donde estuvo X.
     gotoxy(LCol, LRow);
     printf("%c", DeathChar);
     }
     LifeUnit(char ch, int Row_loc, int Col_loc, LifeField *L) {
          LRow = Row_loc;
          LCol = Col_loc;
          LF = L; // Recuerda en que cuadro esta viviendo.
          draw(ch);
          }
      // Limpia antes que desaparesca el objeto
      ~LifeUnit() { erase(); }
      // Ve que todas las condiciones se cumplan
      // para que pueda sobrevivir en el siguiente
```

```
      // ciclo de vida.
      int Alive(); // regresa 1 si vive 0 si murió
};

// Regresa 1 si esta enbarazada.
int Fertile(int Row, int Col, LifeField &LF);
// Función de ayuda para las funciones Fertile() y Alive()
int NeighborAlive(int nr, int nc, LifeField &LF);

#endif  // LIFEUNIT_H_
```

Los metodos de **LifeUnit** incluyen las reglas para Life. En este caso **Alive()** y **Fertile()**, simplemente verifican para ver con cuantos vecinos cuenta la **LifeUnit**, aunque se puede modificar el algoritmo, a uno más complejo.

A continuación se presenta el método de la clase **LifeUnit**.

```
// Programa 64A, Las reglas de la vida. Cada LifeUnit
// calcula si sobrevivirá en la siguiente generación.
#include "plcp64b.h"

const MinimumNeighbors = 2;
const MaximumNeighbors = 3;
const MinimumParents = 3;
const MaximumParents = 3;

// Comprueba si la celda especificada esta ocupada.
int NeighborAlive(int nr, int nc, LifeField &LF)
{
    if(LF.InRange(nr, nc))  // Si no es puntero Nulo
        return( LF(nr, nc) ? 1 : 0); // Usa el operator()()
    return 0;
}

// Cuenta el número de unidades alrededor de R. C.
int SurroundingUnit(int R, int C, LifeField &LF)
{
    return
        NeighborAlive(R-1, C, LF) +
        NeighborAlive(R+1, C, LF) +
        NeighborAlive(R, C+1, LF) +
        NeighborAlive(R, C-1, LF) +
        NeighborAlive(R-1, C-1, LF) +
        NeighborAlive(R+1, C-1, LF) +
        NeighborAlive(R-1, C+1, LF) +
        NeighborAlive(R+1, C+1, LF);
}
```

```
// La función alive() determina si LifeUnit esta vivo
// en la siguiente generación, comprobando cuantas
// de otras unidades estan vivas.

int LifeUnit::Alive()
{
    int Neighbors = SurroundingUnit(LRow, LCol, *LF);
    if( Neighbors >= MinimumNeighbors &&
        Neighbors <= MaximumNeighbors)
            return 1; // Esta vivo
    return 0; // Murió de soledad o de sobrepoblación.
}

// La función Fertile() determina si las condiciones para
// una nueva vida son adecuadas (presumiblemente en una
// celda vacia). Utiliza las mismas reglas que la función
// alive(), pero pueden ser facilmente cambiadas.
int Fertile(int Row, int Col, LifeField &LF)
{
    int Neighbors = SurroundingUnit(Row, Col, LF);
    if( Neighbors >= MinimumParents &&
        Neighbors <= MaximumParents)
        return 1; // Embarazada
    return 0; // también pocos parientes, o sobrepoblación
}
```

En este caso particular, para que una **LifeUnit** viva, deberá contar con 2 o 3 vecinos, de otra manera morira.

Para que se produsca un nacimiento el lugar vacio deberá contar con tres parientes. Número mayor o número menor no produce un nacimiento. La función NeighbordAlive() verifica si hay o no un vecino, la función SurroundingUnit() regresa el número de vecinos o pariente, ya sea para ver si una LifeUnit va a vivir o si se va a producir un nacimiento, respectivamente.

3.e) El lugar donde viven las **LifeUnit.**

Por lo general, el siguiente estado de una simulación, depende del estado presente y de algunos estados anteriores. En este ejemplo el estado siguiente solo depende del estado actual, aunque ésto se puede modificar cambiando el valor de **history**, este valor determina el número de estados, ya que crea un arreglo bidimensional para cada estado. En el presente ejemplo el valor de **history** es de 2, un arreglo bidimensional para el estado actual, y uno para el estado siguiente.

Cada vez que el sistema es operado, se examina el estado actual y se crea el estado siguiente, en base al estado actual.

Si **history** fuera igual a 3, entonces, deberiamos examinar el estado actual, y crear el siguiente estado, en base al estado actual y el estado anterior.

A continuación se presenta la interface para la clase LifeField.

```
// Una clase para almacenar un campo de LifeUnit
// Programa PLCP64B.H
#ifndef LIFEFIELD_H_
#define LIFEFIELD_H_
#include "plcp64a.h"
#include <stdlib.h>

const rowmin = 1;
const rows = 25;
const rowmax = rows - 2;
const colmin = 1;
const cols = 80;
const colmax = cols - 2;

// Se puede tener cualquier cantidad de historia en la simulación
// este ejemplo ve un estado anterior.

const history = 2;

class LifeField {
    LifeUnit *Field[history][rows][cols];
    // Los ciclos del sistema a través de arreglos.
    // La etapa actual de tiempo es definida por la
    // siguiente variable
    int CurrentField; // Selecciona Field[0], Field[1], etc.
    void error(char *msg1 = "", char *msg2 = "") {
      cerr << "LifeField error: " << msg1 << " " << msg2 << endl;
      exit(1);
    }
    void RangeTest(int r, int c, char *msg = "") {
        if(r <= rowmin || r >= rowmax)
            error(msg, "rows out of range");
        if(c <= colmin || c >= colmax)
            error(msg, "cols out of range");
    }
public:
    LifeField(); // inicializa todos los punteros a cero.
                 // (Un puntero cero significa
    // que no hay una LifeUnit en esa localidad.
    const LifeUnit *operator()(int r, int c) {
        RangeTest(r, c, "operator()");
        return Field[CurrentField][r][c];
    }
    // Regresa 0 si esta fuera de rango, 1 de otra forma.
    int InRange(int r, int c);
    void Randomize(int factor); //Pone la LifeUnit de manera
```

```
                                    // aleatoria.
    int NextGeneration(); // Regresa !=0, si LifeUnit permanece
    void Draw_Next_G();  // Dibuja los Neighbords vivos
};

#endif  // LIFEFIELD_H_
```

En realidad el dato miembro **Field** es un puntero a un arreglo tridimensional de punteros, él cual, en este caso particular tiene las siguientes dimensiones **Field[2][25][80]**. Este arreglo de punteros se va a manejar como 2 arreglos bidimensionales, donde uno de los arreglos representa el estado actual, el otro se va llenando con las **LifeUnit** que pasan a la siguiente generación y las **LifeUnit** que van naciendo. En la siguiente generación se invierten los papeles, después de limpiar el arreglo usado en la generación anterior, ahora sirve para ir almacenando las **LifeUnit** que pasaran a la siguiente generación y aquellas que van naciendo.

El dato miembro **CurrentField** almacena el índice del arreglo de punteros que nos define con cual arreglo bidimensional se esta trabajando.

La función miembro **error()** nos dice si ocurrió un error en el manejo del arreglo, basicamente si nos salimos del rango en los indices.

La función RangeTest() es la encargada de la verificación de que los indices se encuentren dentro de los rangos definidos, si por alguna causa, no sucede así, manda llamar a la función error().

La función LifeField(), inicializa todos los punteros a cero. un apuntador, (Un puntero cero significa que no hay una **LifeUnit** en esa localidad.

El **operator()()**, solo verifica que el elemento de **LifeField** que se va a regresar, sea un elemento válido, (indices dentro de rangos válidos), regresa un puntero a **Field**, para una posición [CurrentField][r][c], en el arreglo de punteros. Este operador se usa en la función NeighbordAlive(), para regresar un valor (1) ó (0), ya sea que haya o no haya un vecino. Si lo hay regresa un puntero válido, si no, regresa un CERO.

Para analizar las siguientes funciones, veamos la implementación de la clase LifeField.

```
// Método de la clase LifeField
// Programa PLCP64B.CPP
#include "plcp64b.h"
#include <time.h>

int LifeField::InRange(int r, int c)
{
```

```
    if(r <= rowmin || r >= rowmax)
        return 0;
    if(c <= colmin || c >= colmax)
        return 0;
    return 1;
}

LifeField::LifeField()
{
    // Inicializa todo a CERO
    for(int fld = 0; fld < history; fld++) {
        for(int i = 0; i < rows; i++) {
            for(int j = 0; j < cols; j++)
                Field[fld][i][j] = 0;
        }
    }
    CurrentField = 0;
}

void LifeField::Randomize(int factor)
{
    // Se usa para poner el número aleatorio en 0 o en 1
    // incrementa el divisor por un número aleatorio
    const cutoff = RAND_MAX / factor;
    // Genera el número aleatorio usando el tiempo
    // de la máquina
    time_t tnow;
    time(&tnow);
    srand(tnow);


    for(int r = rowmin; r < rowmax; r++)   {
        for(int c = colmin; c < colmax; c++)
            if ( !(rand() / cutoff) ) {    // división entera
                Field[CurrentField][r][c] =
                            new LifeUnit(LifeChar, r, c, this);
                if( Field[CurrentField][r][c] == 0) {
                    cerr << "new falla en randomize()";
                    exit(1);
                }
            }
    }
}

int LifeField::NextGeneration()
{
    int StillAlive = 0;
    int Change = 0;     // Indica cambio en el estado.
    // Cuenta 0, 1,....history-1
```

```
    int NextField = (CurrentField+1) % history;
    for(int r = rowmin; r < rowmax; r++) {
        for(int c = colmin; c < colmax; c++) {
            // Si hay uno, y esta vivo, lo copia
            if( Field[CurrentField][r][c]) {
             if( Field[CurrentField][r][c] -> Alive()) {
              Field[NextField][r][c] = Field[CurrentField][r][c];
                    StillAlive++;
               } else {
                    delete Field[CurrentField][r][c];
                    // El destructor borra el caracter
                    Field[CurrentField][r][c] = 0;
                    Change++;
               }
            } else { // Uno no esta actualmente aqui
            // Ve si el espacio esta libre para una nueva vida
                if( Fertile(r, c, *this)) {
                    Field[NextField][r][c] =
                         new LifeUnit(birthChar, r, c, this);
                    // Un nacimiento
                    StillAlive++;
                }
            }
        }
    }

for(r = rowmin; r < rowmax; r++)    // Limpia campo actual
    for(int c = colmin; c < colmax; c++)
        Field[CurrentField][r][c] = 0;
CurrentField = NextField; // Avanza en el tiempo
if(!Change)
    return 0;
return StillAlive;
}

void LifeField::Draw_Next_G()
{
    clrscr();
    for(int r = rowmin; r < rowmax; r++)
        for(int c = colmin; c < colmax; c++)
            if(Field[CurrentField][r][c])
                Field[CurrentField][r][c] -> draw(LifeChar);
}
```

Sigamos analizando la clase LifeField.

Primero tenemos la implementación de la función InRange(), la que como dijimos anteriormente nos permite verificar que los indices se encuentren dentro de rangos válidos.

La siguiente función es el constructor de la clase LifeField, él cual nos inicializa el arreglo de punteros a CERO, equivalente en C a NULL. Y también inicializa el dato miembro CurrentField = 0.

La siguiente función es Ramdomize(), la cual recibe como argumento el número dado al llamar al programa en la línea de comandos, mediante el cual y ayudados por la constante definida en stdlib.h RAND_MAX (0x7fff), calculamos un factor (cutoff), el cual, también ayudado por el número seudoaleatorio, generado por la función de la librería rand(), van a definir que **LifeUnit**'s van a ser creadas y en que posición del arreglo de punteros **Field** se va a alojar su dirección, la cual va a ser dada por el operador **new** y va a corresponder a una dirección del heap.

Sintaxis para reservar memoria en el heap para un objeto.

Sintaxis de new.

```
tipo *puntero = new<nom_tipo>(<inicialización>);
```

Donde:

nom_tipo	Puede ser un tipo o una expresión. Ejemplo float *p = new float(x*y);
inicialización.	Una llamada explícita al constructor de la clase. Como es el caso del ejemplo.
puntero	El nombre de una variable puntero.

Sintaxis de delete

```
delete puntero;
```

Como ya mencionamos, cuando se llama al operador **new**, además de solicitar memoria al heap, también manda llamar al constructor de la clase.

El operador **delete**, además de dejar libre la memoria del heap, manda llamar al destructor de la clase, si éste fué definido.

La función **time()** nos regresa el tiempo actual, de acuerdo al reloj de la computadora, en segundos, tomando como referencia Enero 1 de 1970.

Sintaxis: `time_t time(time_t *timer);`

La función Randomize() se basa en el tiempo de la computadora para generar el número aleatorio, para ello usa la función srand(), de la librería del paquete.

En la función Randomize(), solo serán creados aquellos **LifeUnit** para los cuales el valor de rand()/cutoff = 0. Cada **LifeUnit** se va a ir imprimiendo en su posición, conforme va siendo creada (Esto se hace en la llamada al constructor).

La función NextGeneration() realiza cuatro actividades principales.

Mientras va recorriendo todos los elementos de uno de los arreglos bidimensionales.

1.- En cada casilla verifica si hay una **lifeUnit**, ésto lo hace viendo si no hay un CERO en la casilla.

2.- Si la casilla esta ocupada, verifica cuantos vecinos tiene la **LifeUnit**, si no cumple con la regla, es destruida, al liberar la memoria se llama al destructor el cual coloca un CERO en el lugar anteriormente ocupado por la **lifeUnit**.
Si cumple con la regla, es copiada, al arreglo bidimensional contiguo, definido por Field[NextField][r][c].

3.- Si la casilla esta desocupada verifica cuantos parientes hay alrededor de la casilla vacia, si tiene TRES parientes, produce un nacimiento de una nueva **LifeUnit** (En la pantalla este acontecimiento se marca con una S). En este caso también la **lifeUnit** es copiada al arreglo bidimensional contiguo.

4.- Una vez recorrido todo el arreglo, el arreglo bidimensional, él cual fué recorrido es limpiado a CERO, y el arreglo al cual fueron copiadas las **LifeUnit** es definido como el actual, ahora podemos realizar una nueva simulación con las **LifeUnit** que perduraron de la generación anterior y aquellas que nacieron también en la generación anterior.

La función NextGeneration(), regresa el número de **LifeUnit** que pasaron a la siguiente generación, si ninguna paso, la función regresa CERO. Aunque pudo haber habido nacimientos, éstos no son considerados.

Por último, la función Draw_Next_G(), nos manda a la pantalla, las **LifeUnit**, vivas, ésto es, aquellas que siguen viviendo en la nueva generación.

3.f) Corriendo la simulación.

En esta sección presentamos el programa que corre la simulación. Crea un campo para que vivan las **lifeUnit**, llamado: (**LifeField**).

```
// El simulador de vida
// Programa PLCP64X.CPP, Aplicación.
#include "plcp64b.h"

void main(int argc, char *argv[])
{
     if(argc < 2) {
          cerr << "Use: Un entero de vida" << endl
               << "Entre mayor sea el entero"
               << ", menos vidas son creadas en"
               << " el campo de vida";
               exit(1);
     }
     clrscr();
     printf("Para Terminar oprime ESC");
     printf("\nPara continuar oprima cualquier Tecla...");
     getch();

     int i = 1, factor = atoi(argv[1]);
     LifeField Simulation;

     clrscr();
     Simulation.Randomize(factor);
     gotoxy(1, 24);
     printf("Oprima una tecla para continuar...");
     getch();
     while(Simulation.NextGeneration()) {
       char c = 'x';

       gotoxy(1, 24);
       printf("Oprima una tecla para continuar, ESC para Salir");
       gotoxy(78,24);
       printf("%d", i++);
       c = getch();
       if(c == 27) break;
       Simulation.Draw_Next_G();
       gotoxy(1, 24);
       printf("Oprima una tecla para continuar...");
       getch();
     }
}
```

OBSERVACIONES.

Para correr el simulador debemos dar un argumento en la línea de comandos, dicho argumento debe ser un número, se recomienda entre 2 a 10, no más, ya que con el valor de 10, ya se vuelve muy ralo el campo de vida, (se crean pocas **LifeUnit**).

El número dado en la línea de comandos, en realidad es una cadena, por lo que es necesario convertirlo a forma numérica, lo que se hace mediante la función **atoi()**.

Se manda llamar a la función Randomize(), referida por el objeto Simulation. En este punto se crean las **LifeUnit**, al mismo tiempo son dibujadas en la pantalla.

En seguida se llama a la función NextGeneration(), la cual nos define las **LifeUnit**, que permanecieron vivas, descartando las que murieron. Esta función esta dentro de un while, lo que nos permitirá seguir la simulación por varias generaciones, e ir analizando su comportamiento.

El programa va a concluir cuando después de terminar la simulación de una generación oprimamos la tecla ESC o bien, no subsista ninguna de las **lifeUnit** para la siguiente generación.