13

2ej.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

Optimización Basada en Algoritmos Genéticos

un enfoque orientado a objetos

Seminario de Investigación Informática

que en opción al grado de

Licenciado en Informática

presenta:

Juan Raymundo Iglesias León

Tutor:

L.A. y L.C. José Antonio Echenique García

Asesor:

Dr. José Negrete Martínez

México, D.F. 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

A Dios por tantas bendiciones.

RECONOCIMIENTOS

Esta obra, que representa la feliz culminación no solo de una etapa académica sino de mi vida, ha requerido para su realización de la participación de una gran cantidad de personas que de alguna u otra forma han colaborado con sumo entusiasmo y sin más interés que mi propio desarrollo. Por ello, he buscado estas líneas para manifestarles mi agradecimiento de todo corazón y de alguna forma brindarles mi sincero reconocimiento. A mis padres, Imelda y Raymundo, por que desde niño me inculcaron con amor una inmejorable educación, aquí está mi humilde fruto. A Alejandro y Gerardo, por todo el apoyo que como hermanos de verdad me han brindado desde niños, espero nunca fallarles. A todos mis maestros, que me han entregado no solo sus conocimientos sino un poquito de su vida, no defraudaré sus esfuerzos. A mi Universidad, la máxima casa de estudios del país, por brindarme las bases necesarias para poder servir a mi país, a mi gente. A mi tutor, el maestro José Antonio Echenique, por su invaluable respaldo en esta empresa, le estoy en absoluta deuda. A mi asesor, José Negrete, quien me brindó sus conocimientos y entera confianza, todo mi respeto y admiración. A Francisco Cervantes, quién es el responsable del diseño de todo este escrito, gracias por ser verdadera amistad desde niños. A todos mis familiares y amigos, pues es este el resultado de su aceptación y diaria convivencia, espero ser siempre lo que ustedes significan para mi.

CONTENIDO

CONVENIOS

Durante el desarrollo de este trabajo se dan por acordados los siguientes convenios:

- Este tipo de letra será usado para hacer referencia al código de un programa y significa que el texto debe ser introducido como aparece para que tal código funcione.

- **Este tipo de letra será usado durante la explicación de la sintaxis de alguna sentencia y significa que el texto debe ser introducido como aparece para que tal sentencia funcione.**

- *Este tipo de letra será usado durante la explicación de la sintaxis de alguna sentencia y significa que el texto debe ser reemplazado por algún elemento introducido por el programador.*

INTRODUCCIÓN

Los algoritmos genéticos simulan los mecanismos de la genética natural para obtener una técnica de búsqueda que puede aplicarse a muy amplios campos. Su metodología está basada en análisis matemáticos y técnicas computacionales, pero sobre todo en analogías naturales. Los algoritmos genéticos ofrecen un proceso robusto que puede llevar fácilmente a la resolución de problemas a través de la adaptación. Más allá de su intrincado nombre o de su gran habilidad para la búsqueda de soluciones, los algoritmos genéticos se basan en procesos sencillos y sobre todo, computacionalmente posibles. Todo esto ha impulsado a elaborar este manuscrito en el que se presenta un nuevo enfoque para la construcción de algoritmos genéticos, contribuyendo de esta forma con su estudio, pues con respecto a sus procesos intrínsecos planteados por otros autores, se abolen algunas rutinas, se proponen nuevas y se mejoran otras.

Las técnicas de programación orientadas a objetos proporcionan al programador, una forma más cotidiana de resolver los problemas computacionales, además de hacer los códigos más entendibles, flexibles, generalizables y reusables. Por ello se ha elegido dar un enfoque orientado a objetos a nuestra particular concepción de los algoritmos genéticos. Para la simulación orientada a objetos de los algoritmos

genéticos en el computador, se ha preferido la utilización del lenguaje C++, puesto que su código puede ser más rápido comparado con otros lenguajes orientados a objetos, además, este lenguaje es sin duda el más popular para aplicaciones con objetos, por lo que fácilmente se podrá conseguir un compilador con el cual practicar.

El presente trabajo pretende servir como guía de autoestudio, por lo que no es necesario un conocimiento del lenguaje C++, ni de programación orientada a objetos, aún cuando el lector debiera tener preferentemente algunas nociones básicas acerca de programación de computadoras. En el desarrollo del texto, se asume un entendimiento elemental de métodos numéricos, estadística, investigación de operaciones y probabilidad, no se asume un conocimiento genético, por lo que los conceptos al respecto son abarcados con la profundidad considerada como necesaria.

Se ha preparado este documento con un breve repaso de las bases matemáticas requeridas donde se ha creído necesario. También se ha pensado en una sencilla introducción a la programación orientada a objetos y en específico al lenguaje C++, de hecho, los elementos utilizados de éste lenguaje son presentados y explicados paulatinamente mientras se avanza la lectura. Por sobre las matemáticas, los conceptos computacionales o genéticos, se ha tenido la intención principal de mostrar los procesos que permiten que los algoritmos genéticos puedan considerarse como una técnica fuerte a ser utilizada para la resolución de problemas dentro de la informática.

En el capítulo titulado *Algoritmos genéticos*, se presentan las definiciones necesarias para la comprensión de este concepto. Se introducen las bases genéticas en las que se sustentan, su historia y las principales aplicaciones en las que se les ha usado.

El capítulo llamado *Un ejemplo paso a paso*, tiene un solo objetivo: que al final de su lectura se comprenda cómo es que funciona un algoritmo genético sin inmiscuir términos computacionales en lo absoluto.

El capítulo *Enfoque orientado a objetos*, pretende ser una concisa, pero explícita descripción de lo que es el paradigma de la programación orientada a objetos. Al final, ya que se tienen definidos los conceptos básicos, se presenta el plan de trabajo que se seguirá para la construcción de algoritmos genéticos con un enfoque orientado a objetos.

Desde el capítulo *DepUtil.h* hasta el capítulo *Optimización Basada en Genética*, se detalla al por menor la implantación del algoritmo genético propuesto, iniciando desde una programoteca básica hasta un objeto de propósito general capaz de optimizar funciones.

En el capítulo *El viajero* se prueba nuestro algoritmo genético contra un problema de la vida real y que ha sido motivo de estudio en casi cualquier texto de investigación de operaciones y de inteligencia artificial, precisamente: el problema del viajero.

En la mayoría de los capítulos se optó por introducir una sección de ejercicios que mantengan al lector en un caluroso y saludable nivel de entretenimiento. Se encontrarán principalmente dos tipos de ejercicios, los

opcionales y los obligatorios. Los opcionales servirán para reforzar el entendimiento y borrar dudas menores acerca del funcionamiento de algún proceso. Los obligatorios, tratan de ser ejercicios "inteligentes", pues fuerzan al desarrollo de alguna tarea, sin la cuál no se podría avanzar al siguiente capítulo, pues quedaría incompleta alguna rutina de la implantación hecha. Ningún ejercicio obligatorio debiera resultar un imposible, pues se ha cuidado tanto el nivel de dificultad, como el proporcionar con anterioridad las herramientas y pistas necesarias, así como una descripción generosa de lo que se requiere.

Por último, el autor acepta gustoso las observaciones que pudieran enriquecer esta investigación, pues aún cuando se ha tratado con cautela de conseguir objetividad en el texto y eficiencia en los algoritmos, toda invención que fuere hecha por un humano será susceptible de mejorarse mientras existiere otro humano.

J. Raymundo Iglesias L.

1. ALGORITMOS GENÉTICOS

En este capítulo se presentan los elementos para la comprensión del concepto "algoritmo genético", tales como qué es, para qué sirve, de qué consta, de dónde ha surgido la idea de su utilización y en qué se ha utilizado.

1.1. CONCEPTOS GENÉTICOS

Antes de iniciar discusiones mas específicas, es necesario el revisar términos genéticos que se utilizarán intensivamente, y que forman el sustento teórico de la investigación y el desarrollo de los algoritmos genéticos

El conjunto de conocimientos que se tienen acerca de la herencia, las teorías que los explican, la variación biológica y sus consecuencias de orden práctico, constituyen una ciencia que se conoce como genética y que tiene enorme importancia en la biología moderna.

A principios del siglo XX, algunos biólogos como Bateson, Castle, Bridges y otros, expusieron una nueva teoría que ha tenido una brillante confirmación con los hechos a medida que mejor se conocen éstos, la cual se ha denominado teoría genética y que ha sido desarrollada hasta alcanzar su forma más exacta por Thomas H. Morgan, biólogo norteamericano. Esta teoría admite la existencia de entidades o partículas materiales llamadas factores, genas o genes, capaces de segregación independiente, de intercambio entre organismos, y transportar determinados caracteres.

En terminología genética, un cromosoma es un orgánulo autoreproducible, que se halla en el núcleo celular en un número constante para cada especie. Las unidades biológicas de los cromosomas son los genes. El gen es la unidad genética elemental y portadora de los caracteres hereditarios. La individualidad de los genes les permite intercambiarse entre dos cromosomas homólogos, alterar su posición y modificarse, procesos todos que afectan a los caracteres hereditarios.

A la posición guardada por un gen dentro de un cromosoma se le conoce por el nombre de locus. A los valores que puede asumir un gen se les llama alelos. Uno o mas cromosomas se combinan de tal forma que contienen la información genética total acerca de la construcción y operación de un organismo.

Durante el proceso de reproducción se presentan con relativa frecuencia fenómenos cromosómicos interesantes, entre los que se encuentran el entrecruzamiento (o también crossing-over) y la mutación.

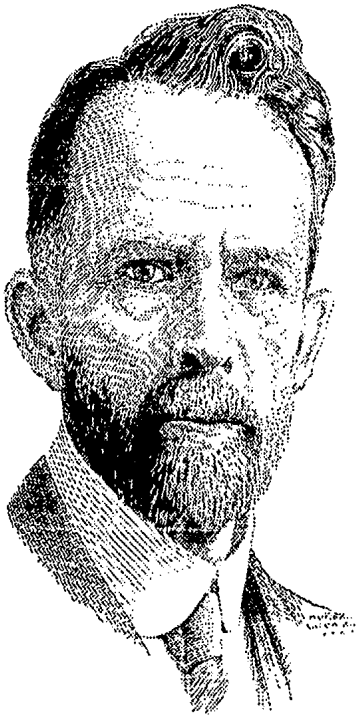

Figura 1.1 Thomas H. Morgan uno de los investigadores que han contribuido a fundar y afirmar la teoría genética de la herencia.

Los genetistas han descubierto que cromosomas alelomorfos pueden en ocasiones entrecruzarse y soldarse de tal modo, que una o más porciones de uno de ellos se intercambian con el alelomorfo

correspondiente, originándose cromosomas mixtos; de acuerdo con lo que la teoría supone, en el nuevo cromosoma así formado aparecerán factores mezclados de uno y otro de los cromosomas primitivos; esto es lo que han llamado entrecruzamiento. Un ejemplo de entrecruzamientos entre cromosomas alelomorfos es mostrados en la Figura 1.2.

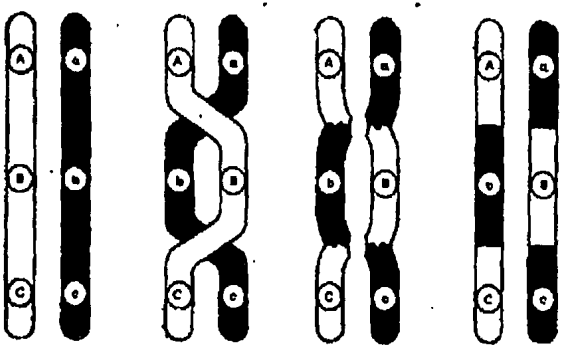

Figura 1.2 Ejemplo de entrecruzamiento cromosómico con doble intercambio de genes

Un ejemplo clásico de entrecruzamiento es el de la mosca del vinagre, Drosophila Melanogaster, estudiada minuciosamente por Morgan. Si se cruza una mosca de cuerpo negro y alas vestigiales, con una mosca de cuerpo gris y alas largas, en la primera generación se obtienen moscas grises de alas largas. Si una hembra de esta generación vuelve a cruzarse con un macho negro y de alas vestigiales, el 82% de los descendientes son negros de alas vegistiales y grises de alas largas y el 18% restante son de alas vegistiales y negros de alas largas, tal y como se muestra en

la Figura 1.3. Estos corresponden a casos de entrecruzamiento cromosómico.

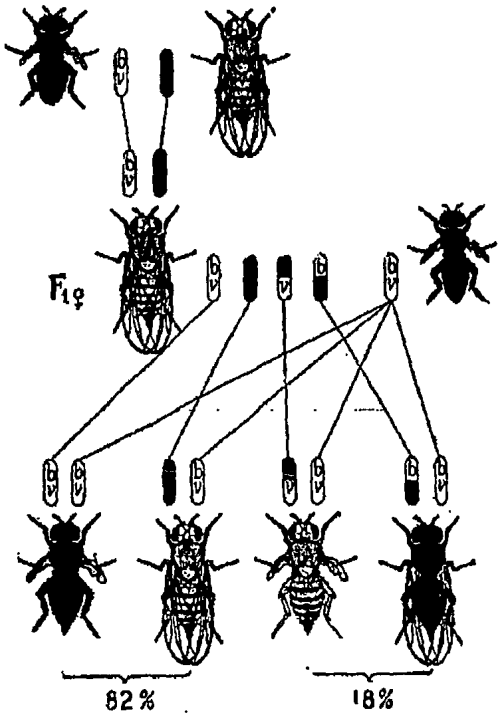

Figura 1.3 Entrecruzamiento en *Drosophila melanogaster*. Representación esquemática según Morgan.

Otro fenómeno cromosómico importante y presentado con menos frecuencia en la naturaleza, es el de la mutación. La mutación se refiere a las alteraciones producidas en la estructura o en el número de los genes o de los cromosomas de un organismo vivo, que se transmiten a los descendientes por herencia.

Cada fenómeno cromosómico, tal como el entrecruzamiento o la mutación, pueden representar para una especie un resultado letal, una deficiencia, o bien, una mejoría. En este último caso, se trataría de una ventaja para la adaptación de esa especie, por lo que los individuos con esa mutación prevalecen, es decir, se seleccionan. Son pues los fenómenos cromosómicos, un factor importante en la evolución, dado que es un proceso que bien pudiera significar la supervivencia para una especie.

A la información genética total, es decir, la reunión total de genes que constituyen el patrimonio hereditario de un individuo, se le llama genotipo. El organismo formado por la interacción del genotipo con su ambiente, es llamado fenotipo, en otras palabras, el fenotipo es el conjunto de caracteres exteriores presentados por un individuo.

1.2. DEFINICIÓN

Desde sus inicios, uno de los tópicos de investigación en el campo de la inteligencia artificial que más estudio ha acaparado, es sin duda el de los métodos de búsqueda de soluciones para resolución de problemas. De

hecho, la capacidad de resolver problemas suele usarse como una medida de la inteligencia, tanto para un hombre como para una máquina.

Un algritmo genético es un modelo de búsqueda de soluciones para resolución de problemas basado en la simulación de los procesos genéticos. El algoritmo genético incorpora dichos procesos para emular una población de seres, cada uno de los cuales contiene información que lleva a la solución de un problema en particular. La población es expuesta a un mecanismo de selección, en el que los individuos más fuertes, esto es, los que han mostrado mayor eficiencia al resolver un problema, tienen mayor probabilidad de acceso a métodos de supervivencia y reproducción, mientras que a los a habitantes mas débiles e ineficientes les espera la muerte. El éxito mostrado por un algoritmo genético al resolver problemas, radica en la utilización de la información histórica registrada en la fuerza de cada ser y en el paso de una generación a otra para crear nuevas formas de solución a través de la reproducción.

A Jhon Holland de la Universidad de Michigan y colaboradores se deben las primeras ideas acerca de los algoritmos genéticos. Las metas que dieron objeto a su investigación fueron principalmente, la abstracción del proceso de adaptación de los sistemas naturales y el diseño de sistemas artificiales dotados de los mecanismos que tienen dichos sistemas naturales. Las primera documentación en el tema fue realizada en 1975, bajo el nombre de *A daption of Natural and Artificial Systems*, y se debe precisamente a Holland. Desde entonces los algoritmos genéticos ha sido motivo de creciente investigación así como de diversas publicaciones, entre estas, una de las mejores es la de Goldsberg en su *Genetic Algorithms in Search, Optimization, and Machine Learning.*

1.3. PROCESOS BÁSICOS

Revisemos ahora, una serie de procesos genéticos que aplicados a una implantación computacional permiten la resolución de problemas: la reproducción, la selección y la adaptación.

La reproducción en un algoritmo genético es el proceso que trae como resultado la creación de una generación, en la que nace un nuevo conjunto de conocimientos. Durante la reproducción de un algoritmo genético pueden simularse los fenómenos cromosómicos que se presentan en la naturaleza tales como el entrecruzamiento y la mutación. Los fenómenos cromosómicos juegan un papel importante en el funcionamiento de un algoritmo genético pues en ocasiones es conveniente la creación de cromosomas que provean de algún cambio adaptativo para probarse como una nueva forma de solución. La propiedad de reproducción de una algoritmo genético tiene la cualidad de aparentar el sentido de la intuición e innovación humana para resolver problemas al generar nuevos seres que puedan aportar novadoras formas de solución a un problema.

La selección en un algoritmo genético es el proceso a través del cual se logra la elección de los individuos destinados a reproducirse o a morir para conseguir mejoras en la población. La selección se sustenta en bases probabilísticas de acuerdo a la fuerza desarrollada por cada individuo. Entre más fuerte sea un ser, mayor será su probabilidad de reproducirse y menor su probabilidad de morir. La fuerza de un individuo se incrementa a medida que su información para resolver un problema sea

mejor, esto corresponde, en términos biológicos a la adecuación mostrada por un individuo ante su entorno. Mediante el mecanismo de selección, generación a generación se asegura que el sistema artificial funcione mejor.

El proceso de adaptación de un algoritmo genético es el simple resultado de los procesos de selección y reproducción. Ha sido también su central cualidad, pues la adaptación le permite su supervivencia en diversos ambientes, y no solo eso, sino que también reduce o evita lo costoso del rediseño de un sistema. Si un problema cambiara, ya sea en alguna de sus componentes o en su totalidad, el algoritmo genético hará que la base de conocimientos cambie de acuerdo al nuevo enfoque del problema. A través de la simulación de los sistemas biológicos de adaptación natural, el sistema artificial conservará su vigencia con el paso del tiempo.

La adaptación al medio representa implícitamente un continuo autoaprendizaje y automodificación acorde a las condiciones presentadas por el exterior. La mejor prueba de que la superviviencia de un sistema depende de su capacidad de adaptación, nos la da la naturaleza. El algoritmo genético es adaptativo persé.

1.4. APLICACIONES

La aplicación de un algoritmo genético es tan variada como las áreas en las que se requiera de un sistema de búsqueda de soluciones verdaderamente eficiente y robusto. La implantación de un algoritmo genético es computacionalmente simple, y su uso no está restringido a espacios de búsqueda específicos. Los algoritmos genéticos han tenido históricamente dos usos principales, como método para optimización de funciones y como método para aprendizaje de máquinas.

Algunos de los usos que se han dado a los algoritmos genéticos como un método para optimización de funciones, son el de registro de imágenes médicas, teoría de juegos, investigación de operaciones, investigación y simulación de procesos biológicos, administración, estudios ingenieriles, controladores de tiempo real, solución de ecuaciones no lineales, reconocimiento de patrones y simulación de adaptación en modelos reactivos prehistóricos cazador-presa.

No menos son las aplicaciones existentes para el aprendizaje de máquinas: sistemas para diagnóstico médico, predicción del rendimiento de empresas, aprendizaje de reglas para la descripción de las preferencias de un consumidor, juegos, aprendizaje de las reglas de multiplicación, descripción de sistemas evolutivos, aplicaciones de tiempo real, estudios ingenieriles, robótica, predicción de eventos internacionales y simulación de poblaciones de seres en ambientes de segunda dimensión.

2. UN EJEMPLO PASO A PASO

El propósito de este capítulo es el mostrar el funcionamiento de un algoritmo genético por medio de una especie de prueba de escritorio, en la que, sin inmiscuirnos en problemas computacionales, se desvanezcan las inquietudes de qué es lo que hace un algoritmo genético, de tal forma, que quede en suspenso el cómo se hace para las secciones posteriores.

2.1. POBLACIÓN INICIAL

El mecanismo de un algoritmo genético es extremadamente sencillo y se remite simplemente a la manipulación de secuencias de caracteres. Una secuencia de caracteres es cualquier sucesión de símbolos. Algunos ejemplos de estas secuencias de caracteres son:

- AFJSDL9427MSKJA
- 293048577¿7!"°°@6¬h
- 92
- 01000100010101001
- 01010111

Las dos últimas secuencias son secuencias binarias, puesto que están compuestas por solo dos dígitos (0, 1); son este tipo de secuencias en especial, con las que se trabajará preferentemente.

Lo único que debe hacerse con las secuencias de caracteres es realizar operaciones de copia entre una y otra o de intercambio de caracteres, claro, siguiendo ciertas reglas que no tienen mayor complicación. Esta combinación entre sencillez de programación y potencia en la búsqueda es un atractivo más para el desarrollo de algoritmos genéticos.

Para mostrar el funcionamiento de un algoritmo genético se realizará un sencillo ejemplo en el que revisaremos los procesos más frecuentemente usados. El primer paso es crear una generación aleatoria inicial. Pensemos en una población de 5 seres que están conformados por un solo cromosoma y 6 genes. El cromosoma será binario, es decir, sus alelos podrán ser 1 o 0. La estructura del cromosoma para un individuo es mostrado en la Figura 2.1, mientras que la población inicial generada puede observarse en la Tabla 2.1. El alelo para cada gen en el cromosoma es obtenido en forma aleatoria mediante experimentos binomiales sucesivos, esto quiere decir que para cada individuo en la población se realizaron 6 experimentos binomiales, uno para obtener el alelo de cada gen.

Población	
Individuo	Cromosoma
1	010110
2	101001
3	111000
4	000111
5	010101

Tabla 2.1 Población inicial de individuos con sus respectivos cromosomas.

2.2. REPRODUCCIÓN

Teniendo formada una población inicial, el siguiente paso es crear nuevos seres a través de la reproducción. La reproducción es un proceso por el cual los habitantes probabilísticamente más fuertes procrean a otros seres. Para poder reproducir a los individuos de una población, es necesario, seleccionar antes de manera probabilística a dos seres de los más fuertes de la población.

La operación de selección es una versión artificial que simula la selección natural explicada por Darwin, en la que los individuos más fuertes tienen más probabilidad de ser escogidos para reproducirse. En las

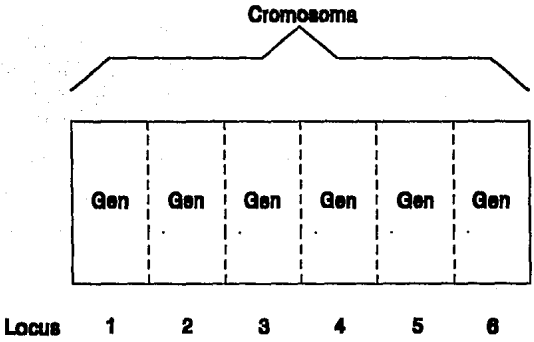

Figura 2.1 Estructura de un cromosoma con seis genes indicándose en la parte inferior el locus de cada gen en el cromosoma

poblaciones naturales, la fuerza de un individuo es determinada principalmente por su habilidad de adecuación ante su ecosistema. En las poblaciones artificiales de un algoritmo genético, la fuerza de un individuo será determinada por su habilidad para resolver un problema, por lo tanto, es necesaria una función objetivo que nos permita saber cuál ha sido el desempeño de cada individuo para un problema en específico.

La función objetivo para obtener la fuerza, varía dependiendo de la naturaleza del problema que se intenta resolver. Para nuestra sencilla simulación veremos en el cromosoma un número en sistema binario y utilizaremos su valor en sistema decimal para darle una fuerza inicial al individuo[1]. El resultado puede observase en la tercera columna de la

[1]Para una explicación más detallada acerca de cómo convertir un número de decimal a binario refiérase a la sección 5.8

Tabla 2.2, donde además en la cuarta columna se proporciona el porcentaje de la fuerza de cada individuo con respecto a la fuerza total.

Población			
Individuo	Cromosoma	Fuerza	%de la fuerza total
1	010110	22	14.96
2	101001	41	27.89
3	111000	56	38.09
4	000111	7	4.76
5	010101	21	14.28
Total		147	100
Promedio		29	20
Máximo		56	38.09

Tabla 2.2 Cálculo de la fuerza para cada individuo de la población

Tal vez la forma más fácil y comúnmente usada para realizar la selección probabilística de los más fuertes es el método de la ruleta cargada. Este método consiste en el simular el juego de la ruleta, en la que cada individuo tiene un porcentaje ganar proporcional al porcentaje de su fuerza con respecto a la fuerza total. Cada vez que se requiera elegir un individuo, se echa a andar la ruleta entre los candidatos para seleccionar a uno.

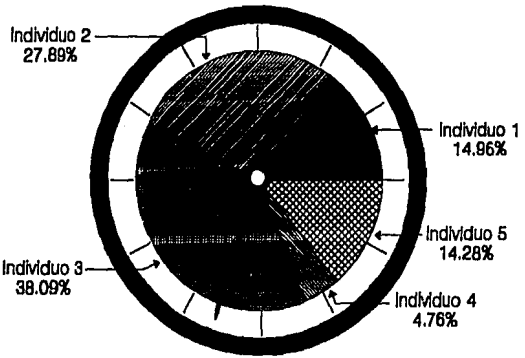

Figura 2.2 Selección probabilística del más fuerte mediante el método de la ruleta cargada.

Para la población de este ejemplo, que se muestra en la Tabla 2.2, la fuerza total en la población, que es la suma de las fuerzas individuales, es de 147, mientras que la fuerza individual máxima es de 56, lo que representa el 38.09% de la fuerza total. La distribución de la ruleta de selección se ejemplifica en la Figura 2.2. Como resultado, las probabilidades de selección para cada individuo serían proporcionales a su fuerza con respecto a la fuerza total, tal y como se puede observar en la Tabla 2.3. Por ejemplo para la fuerza individual máxima, que es 56, existirá una probabilidad de selección de .3809 que es proporcional al valor de su fuerza con respecto a la fuerza total, que es 38.09%. De esta forma, los individuos con mayor fuerza tendrán mayor probabilidad de selección para reproducirse.

Población				
Individuo	Cromosoma	Fuerza	%de la fuerza total	Probabilidad de selección
1	010110	22	14.96	.1496
2	101001	41	27.89	.2789
3	111000	56	38.09	.3809
4	000111	7	4.76	.476
5	010101	21	14.28	.1428
Total		147	100	1.00
Promedio		29	20	.20
Máximo		56	38.09	.3809

Tabla 2.3 Cálculo de la probabilidad de selección para cada individuo de la población

En nuestra población ejemplo, después de haber lanzado la ruleta en dos ocasiones, se han seleccionado los individuos 2 y 5 como padres para reproducirse, esto es mostrado en la Tabla 2.4. El lector enseguida notará con cierta extrañeza que no se haya seleccionado al individuo 3 que representa al más fuerte de la población, pero recuerde que tal como sucede en un sistema natural, el factor aleatorio juega un papel fundamental, y en este caso al individuo 3 la "suerte" no le ha sido favorable. Sin embargo los individuos seleccionados ocupan buenos lugares por su fuerza, por lo que el resultado a fin de cuentas no será malo.

Población		
Individuo	Cromosoma	Seleccionados como Padres
1	010110	
2	101001	Padre1
3	111000	
4	000111	
5	010101	Padre2

Tabla 2.4 Individuos seleccionados aleatoriamente para reproducirse, de acuerdo con el método de la ruleta cargada

Una vez seleccionados los individuos que se reproducirán, se realiza una copia de sus cromosomas, mismas que pudieran verse afectadas por fenómenos cromosómicos tales como el entrecruzamiento y la mutación.

La operación de entrecruzamiento consiste en dos pasos. El primero es seleccionar aleatoriamente el locus de entrecruzamiento que puede ser cualquier locus entre 1 y la longitud del cromosoma menos 1. El segundo es intercambiar los genes de los cromosomas paternos de forma cruzada. La operación de entrecruzamiento se repite dos veces para reproducir dos nuevos individuos hijos, el primer entrecruzamiento copia primero los genes del padre 1 y el segundo copia primero los genes del padre 2, de tal forma que los padres dan origen a dos hijos uno cuyo cromosoma inicia con genes del padre 1 y otro cuyo cromosoma inicia con genes del

padre 2. El operador de entrecruzamiento pude ser visto gráficamente en la Figura 2.3.

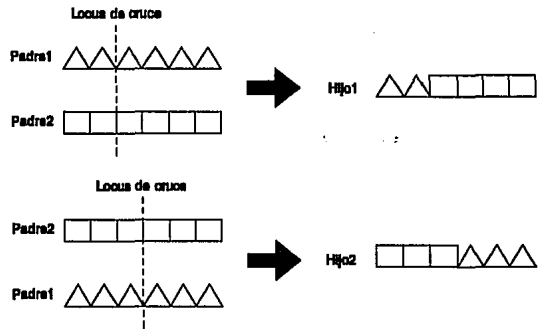

Figura 2.3 Funcionamiento gráfico del operador de entrecruzamiento

El funcionamiento del entrecruzamiento para los padres seleccionados de la población ejemplo se encuentra resumido en la Tabla 2.5. En los dos primeros renglones se observan al Padre1 y al Padre2 con sus respectivos cromosomas. Primero se elige de forma aleatoria al locus de cruce que resulta ser el locus uno, que se muestra en la tercera columna, así como en la segunda pues se añade una barra vertical (|) en el locus de cruce de los dos cromosomas. Posteriormente se copian los cromosomas en forma cruzada, lo que da origen a un hijo cuyo cromosoma está formado por, primero un gen del Padre1 (puesto que el locus de cruce es uno) y después por cinco genes de Padre2 (que es el número de genes restantes a partir del locus 1). Observe que el segundo hijo es el resultado del mismo proceso que originó al primero, sin embargo en esta ocasión

son copiados primero los genes del Padre2 y posteriormente los del Padre1

Padres				
Individuo	Cromosoma	Locus de cruce	Hijo	Fuerza
2(Padre1)	1|01001	1	1|10101	53
5(Padre2)	0|10101			
5(Padre2)	01|0101	2	01|1001	25
2(Padre1)	10|1001			

Tabla 2.5 Entrecruzamiento para los padres seleccionados de la población

Otro fenómeno cromosómico presentado en la naturaleza, la mutación, produce en ocasiones individuos que son verdaderos "monstruos" con deficiencias extremas y pocas posibilidades de supervivencia, pero en otras produce individuos con ventajas adaptativas, que a través del tiempo producen mejorías en la raza. Un ejemplo real de la naturaleza es la teoría de que los insectos consiguieron alas por medio de una mutación de sus antenas. En los sistemas artificiales como en los naturales, la mutación permite buscar mejoras en la población a través de intentos aleatorios.

La operación de mutación en un algoritmo genético consiste en un proceso realizado gen por gen que se efectúa simplemente cambiando el alelo de un gen. Por ejemplo, si el alelo de un gen es 1 y se muta entonces su nuevo alelo será 0.

Hijos				
Individuo	Cromosoma	Locus de mutación	Cromosoma	Fuerza
Hijo1	110101	0	110101	53
Hijo2	011001	4	011101	29

Tabla 2.6 Proceso de mutación en los hijos

Supongamos que para la población ejemplo se tiene una probabilidad de mutación de .08. Siendo que la mutación se aplicará sobre los dos cromosomas de los hijos creados y que cada cromosoma tiene seis genes, entonces son esperadas .08x6x2=.96 mutaciones, es decir, es esperada una mutación para nuestro ejemplo. Efectivamente, tras 12 experimentos se da un proceso de mutación en el gen localizado en el locus 4 del Hijo2 cuyo alelo era 0 y es convertido a 1 como producto de una mutación. Los hijos tras el proceso de mutación son mostrados en la Tabla 2.6.

2.3. MUERTE

El siguiente paso es incorporar a los nuevos individuos a la población, pero siendo seis el tamaño máximo de la población es entonces necesario el que "mueran" dos individuos para que los recién nacidos entren en la población. La operación de muerte consta de dos pasos, primero

seleccionar al individuo que va a morir, segundo desechar ese individuo seleccionado para morir reemplazándolo por uno nuevo.

La selección del individuo que va a morir requiere de un proceso que elija a un ser de entre los más débiles y parecidos al nuevo ser. Así el operador de muerte cumple con varios propósitos, pues al seleccionar al más débil elimina a los individuos que peor desempeño han tenido, mientras que al mismo tiempo cumple con el objetivo de mantener la diversidad en la población. Lograr una gran variedad de individuos es importante para responder a la gran variedad de problemas a la que esté expuesta la población.

Una explicación del algoritmo usado para la selección de los individuos que morirán es pospuesta para la sección 11.5, en el que se expondrá en detalle el algoritmo que propicia la diversidad junto con el método de muestreo estadístico propuesto para selección aleatoria de los individuos más débiles.

Para nuestro ejemplo nos contentaremos con ver por el momento a la muerte como una caja negra, que ha seleccionado para morir al individuo 4 y al 5 (los más débiles de la población), mismos que son sustituidos por los hijos de reciente creación. La nueva población se muestra en la Tabla 2.7. En las tres primeras columnas se presenta a la población anterior, mientras que en las últimas tres están ya reemplazados los individuos 4 y 5 por los hijos 1 y 2. Como puede observarse en las columnas de fuerza, el resultado obvio del algoritmo genético que se ha aplicado durante toda esta sección ha originado que la fuerza total de la siguiente generación se ha incrementado de 147 a 201, la fuerza promedio

se ha incrementado de 29 a 40.2, aún cuando la fuerza máxima sigue siendo de 56.

Población vieja			Población nueva		
Individuo	Cromosoma	Fuerza	Individuo	Cromosoma	Fuerza
1	010110	22	1	010110	22
2	101001	41	2	101001	41
3	111000	56	3	111000	56
4	000111	7	4 (hijo1)	110101	53
5	010101	21	5 (hijo 2)	011101	29
Total		147			201
Promedio		29			40.2
Máximo		56			56

Tabla 2.7 Nueva población

El proceso paso a paso explicado en esta sección dejará ideas más claras acerca como funciona un algoritmo genético, así como también sentará las bases para un mejor entendimiento de los capítulos que restan. Las operaciones genéticas básicas que se han explicado traen como consecuencia la robustez de los algoritmos genéticos como método de búsqueda, así como la apariencia de innovación o creatividad humana, esto es porque después de un proceso genético aparecen nuevos individuos con nuevas formas de resolver problemas. Es en estas

operaciones donde radica la potencia de procesamiento de un algoritmo genético.

3. ENFOQUE ORIENTADO A OBJETOS

Se ha optado por un concreto, pero sustancioso recorrido teórico por el modelado de programas orientado a objetos, para proporcionar los conceptos que permitan exponer, al final del capítulo, el plan de trabajo que se seguirá en este texto para conseguir la construcción de algoritmos genéticos.

Quizá este capítulo resulte muy abstracto en su primera lectura por la gran cantidad de definiciones que son manejadas, por lo que sería muy recomendable una segunda lectura una vez que se haya concluido el estudio del último capítulo de este trabajo.

3.1. DEFINICIÓN

La programación orientada a objetos es un paradigma de la ingeniería de software, cuya filosofía es un modelado de software basado en objetos del mundo real. La programación orientada a objetos puede ser vista

también, como una forma diferente de organización de programas que facilita conceptualizar el mundo real mientras se desarrolla una aplicación. Por ello la programación orientada a objetos puede reducir significativamente la complejidad del problema a comparación de otras técnicas anteriores como la programación estructurada.

La programación orientada a objetos facilita el mantenimiento de un programa al organizarlo en unidades relativamente independientes. Además la programación orientada a objetos permite una mejor reusabilidad del código escrito a través de la herencia de un objeto a otro.

La programación orientada a objetos no viene a reemplazar las diversas prácticas y buenas costumbres programáticas, tales como las de la programación estructurada, sino que al contrario, las usa llevándolas a una concepción más elevada. En la Tabla 3.1 se encuentra un cuadro comparativo de la programación estructurada contra la programación orientada a objetos, de esta forma se pueden ver las principales variaciones conceptuales de una técnica de programación a la otra.

3.2. OBJETOS

Un objeto es un tipo de dato complejo formado por funciones y otros datos. Los datos del objeto representan las características que lo definen, mientras que las funciones que manipulan los datos representan las acciones que puede efectuar dicho objeto. Es frecuente encontrar bibliografía que a los datos de un objeto los llame características, datos

Programación estructurada	Programación Orientada a Objetos
Un programa es una secuencia de pasos que hay que hacer para resolver un problema	Un programa es un conjunto de objetos con los que se puede resolver un problema
Se escribe un procedimiento (o función) por cada tarea que se debe realizar	Se escriben las acciones que ejecutará cada objeto para resolver las tareas que se deben realizar
Los procesos son el factor principal	Los datos son el factor principal
Los procesos guían el desarrollo de la aplicación	Los datos guían el desarrollo de una aplicación
Los procesos dictan el tipo de datos que debe usarse	Los datos dictan los procesos que deben realizarse
El diseño de la aplicación sigue un modelo de arriba-hacia-abajo	El diseño de la aplicación sigue un modelo de abajo-hacia-arriba
Un problema se resuelve subdividiéndolo en otros problemas más pequeños y estos a su vez en otros más pequeños y así sucesivamente hasta solucionar el problema.	Un problema se resuelve con objetos muy genéricos que sirven para crear otros más especializados en el problema, estos a su vez sirven para crear otros objetos más especializados y así sucesivamente hasta solucionar el problema

Tabla 3.1 Cuadro comparativo entre programación orientada a objetos y programación estructurada

miembro ó variables miembro; también que a las funciones de un objeto se les llame acciones, métodos, o funciones miembro. Cualquiera de estos nombres para los datos y funciones de un objeto es bastante bueno y descriptivo, por lo que en adelante se usa cualquiera de ellos

indistintamente. Al conjunto de datos y funciones de un objeto se les conoce como miembros o propiedades de un objeto.

A los argumentos requeridos por un objeto para ejecutar una acción se les llama mensajes. A la secuencia requerida de mensajes para ejecutar la acción de un objeto se le llama protocolo. Por ello, muchos teóricos afirman que "un objeto se comunica con el exterior mediante de un protocolo que es marcado por el paso de mensajes a través de sus acciones".

Si en este momento tantos conceptos lo tienen desconcertado, lo mejor es no perderse en abstracciones de las abstracciones, lo importante es que un objeto puede utilizarse fácilmente en un programa como cualquier otro tipo de dato mediante rutinas que le son definidas. Lo fascinante de la teoría orientada a objetos es mejor en la práctica y pronto habrá mucha práctica.

3.3. PROPIEDADES

Ciertas propiedades deben estar presentes de alguna u otra forma en un lenguaje de programación que se digna de ser orientado a objetos y son el ocultamiento de la información, la construcción, la destrucción, la herencia y el polimorfismo.

Ocultamiento de la información es la propiedad que evita que las características intrínsecas de un objeto puedan ser modificadas directamente por el exterior, de esta forma, un objeto podrá ser utilizado

de manera simplificada mediante acciones que son públicas, es decir accesibles a todos. La abstracción de datos es un concepto muy relacionado con el ocultamiento de la información y se refiere a la utilización de tipos complejos de datos sin importar la forma en que ha sido definida su estructura interna, ni el cómo es que realiza sus procesos. En cierta manera, existe abstracción de datos cuando es posible el ocultamiento de la información, pues al quedar oculta la estructura de un objeto, nos ocupamos entonces de hacer funcionar éste objeto, en lugar de preocuparnos en cómo es que funciona.

La herencia es la definición de una objeto en términos de otro(s). La herencia permite crear una relación jerárquica de padres e hijos entre los objetos, en las que los objetos hijos asumen todas las características y acciones de sus padres. Otro nombre con el que se conoce a la herencia, es el de derivación, por lo que se dice, que una clase puede ser derivada (hija) de una o varias clases base (padres).

El polimorfismo es la propiedad que permite que un objeto responda diferente a una misma acción dependiendo del mensaje que le es enviado. El polimorfismo es posible porque pueden existir cualquier cantidad de funciones con el mismo nombre, siempre y cuando se diferencien en el tipo o número de parámetros o por el tipo de dato retornado. Por estas características al polimorfismo también se le ha conocido como sobrecarga de una función. El polimorfismo puede presentarse desde nivel global, es decir, accesible a cualquier parte de programa, hasta el nivel de operadores, tales como la suma, resta o similares.

Un constructor es una acción asociada a un objeto y que es ejecutada al momento de creación de dicho objeto. En los lenguajes orientados a objetos, generalmente existe un constructor por omisión para cada objeto, tal constructor es el encargado de crear los datos usados por el objeto. Regularmente la definición de un constructor no es obligatoria y basta con el constructor por omisión, pero existen otras ocasiones en las que sería deseable inicializar algunas variables o realizar un proceso previo a la utilización del objeto, como por ejemplo, en objetos que usan memoria dinámica, en tales casos el uso de un constructor puede ser muy útil.

Un destructor es una acción asociada a un objeto y que es ejecutada al momento de destrucción de dicho objeto. El funcionamiento de un destructor es análogo al del constructor. Existe un destructor por omisión cuyo objetivo es el liberar la memoria ocupada por los datos del objeto. Generalmente la definición de un destructor no es necesaria y basta con el destructor por omisión, pero existen ocasiones en las que se desearía hacer un proceso previo a la liberación de la memoria ocupada por los datos, como por ejemplo, en objetos que usan memoria dinámica, en tales casos el uso de un destructor puede convertirse en algo casi obligatorio.

3.4. EL PLAN DE TRABAJO

El plan de trabajo a seguir en las siguientes secciones para la construcción de algoritmos genéticos es presentado en el Figura 3.1, se trata del un desarrollo orientado a objetos basado en una serie de utilerías. Todo con el objetivo de crear un algoritmo genético de propósito general que pueda ser usado fácilmente por una aplicación final.

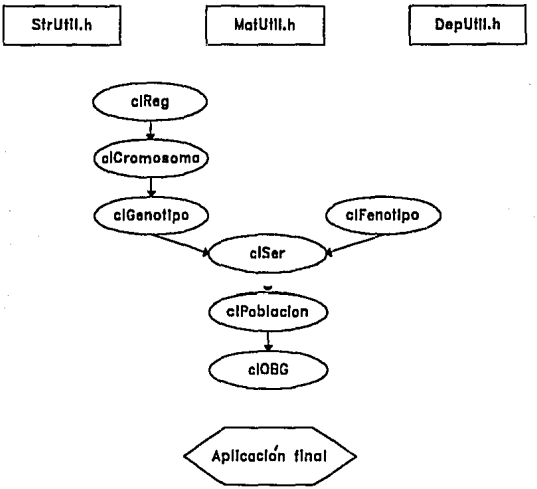

Figura 3.1 Plan de trabajo propuesto para el desarrollo de algoritmos genéticos orientados a objetos

Primero se desarrollarán tres utilerías que conforman una programoteca lo suficientemente completa como para proveer de las herramientas de codificación tanto del algoritmo genético como de la aplicación final. Las tres utilerías se muestran en la Figura 3.1 en forma de cuadrado y son llamadas `StrUtil.h`, `MatUtil.h` y `DepUtil.h`. La utilería `StrUtil.h` contendrá varias rutinas comunes para el manejo de cadenas de caracteres. La utilería `MatUtil.h` servirá para ejecutar varias rutinas matemáticas necesarias para simular diversas situaciones naturales dentro de un algoritmo genético. La utilería `DepUtil.h` servirá para conjutar métodos de

lectura, escritura y presentación de datos para observar el comportamiento del algoritmo genético.

Después se crearán el algoritmo genético y el motor que los ejecuta a través de los objetos clReg, clCromosoma, clGenotipo, clFenotipo, clSer, clPoblacion y clOBG. En la Figura 3.1 se muestran los objetos encerrados en círculos, mientras que con flechas se muestran las dependencias que existen entre ellos. El objeto de tipo clReg, servirá como registro para almacenar datos. El objeto de tipo clCromosoma es una clase hija de clReg que servirá como registro para almacenar genes. El objeto clGenotipo, está formado por varios objetos clCromosoma, para simular el paquete genético que se presenta en los sistemas naturales. El objeto clFenotipo contiene el grado de adecuación que ha conseguido un ser en su ambiente. El objeto clSer está formado por un clGenotipo y un clFenotipo a manera de como está formado un ser en un sistema natural. El objeto clPoblacion está formado por una serie de objetos clSer, y tiene capacidad de realizar las operaciones de reproducción, muerte, selección así como otras más que serán mostradas en su tiempo. El objeto clOBG toma sus siglas de Optimización Basada en Genética, pues su objetivo es precisamente optimizar una función cualquiera a través de un algoritmo genético.

Por último, tenemos a la aplicación final mostrada como un rombo en la Figura 3.1. La aplicación final es un problema del mundo real que se desea resolver a través de la definición de una función objetivo y de llamadas al objeto clOBG. En específico, la aplicación final que abordaremos será la del problema viajero, que se a vuelto un clásico en la literatura de optimización dentro de la investigación de operaciones, así como también para las áreas computacionales y en particular de la inteligencia artificial. El objetivo de presentar la solución al problema del

viajero, es el demostrar que cualquier función que se desee optimizar, se puede optimizar mediante un algoritmo genético y en especial con la estructura de solución genérica basada en objetos que aquí se propone.

4. DepUtil.h

Antes de comenzar la codificación de un Algoritmo Genético, se hace necesario el introducir nociones del lenguaje de programación C++. Para ello, se creará una serie de rutinas de propósito general, que está pensada además para ahorrar tiempo desarrollo de un algoritmo genético. Así, al final de esta sección, el lector tendrá en sus manos una utilería para la depuración de programas a la que nombraremos `DepUtil.h`

4.1. Detener

En C++, como en otros muchos lenguajes de programación, una función es un conjunto de instrucciones que realizan una tarea específica. Una función debe seguir la sintaxis:

```
inline Tipo NombreFunción ( Parámetros )
{
 Instrucciones ;
}
```

Al ejecutarse un programa, el código de una función se encuentra en alguna parte de la memoria de la computadora, y en cualquier momento se puede mandar ejecutar. Este llamado de ejecución requiere consumir algún tiempo de procesador para invocar a la función, pues es necesario localizarla en memoria, preparar entonces la pila del programa, crear después las variables que contiene la función, y finalmente ejecutarla. Pero cuando una función inicia con la palabra reservada[2] `inline`, se indica al compilador de C++, que incruste el código de la función donde sea invocada, es decir, reemplaza la llamada por todo el código de la función. Una función `inline` ofrece la ventaja de hacer el código más rápido, pues ahorra el tiempo consumido en las operaciones antes indicadas para hacer el llamado a la función, pero el precio de la rapidez en el código es el que el programa ejecutable tenga un tamaño mayor, esto es obvio por los reemplazos de código cada vez que es invocada una función. El uso de `inline` es opcional, por ello es responsabilidad del programador el mantener en balace la rapidez y tamaño del código para que un programa ni sea lento, ni sea tan grande que no pueda caber en la memoria con la que se dispone en el computador. Es recomendable utilizar el criterio de nombrar `inline` a las funciones pequeñas o muy ejecutadas, de tal forma que no hagan crecer mucho el código y optimicen la velocidad del mismo.

El *Tipo* es el tipo de un valor que será devuelto por la función al terminar su ejecución. En la Tabla 4.1 se muestra en la primera columna los tipos de datos básicos que puede retornar cualquier función, mientras que en la segunda explica brevemente el uso de cada uno de ellos. El

[2]Una palabra reservada es aquella que tiene un significado predefinido en un lenguaje de programación

especificar el *Tipo* es opcional y en caso de no indicarse se asume por omisión que el tipo del valor retornado es int. Por otra parte, si finalmente la función no retornara valor alguno, entonces puede utilizar void como *Tipo*, lo que significa que la función retornará ningún valor.

TIPO	USO
char	caracter con signo
unsigned char	caracter sin signo
short	entero corto
unsigned short	entero corto sin signo
int	entero
unsigned	entero sin signo
long	entero largo
unsigned long	entero largo sin signo
float	número de punto flotante
double	número de punto flotante con doble presición
long double	número de punto flotante largo de doble presición

Tabla 4.1 Tipos de datos existentes en C++ y su significado

El *NombreFunción* es un identificador con el que se podrá hacer posterior referencia a esa función. Un identificador debe iniciar siempre por un

caracter alfabético[3] o por el caracter de subrayado[4], y puede continuar con cualquier secuencia de caracteres alfanuméricos o el caracter de subrayado. Los siguientes son ejemplos de identificadores válidos:

```
_DepUtil
apMensaje
Detener
```

Es importante señalar que el compilador C++ trata como diferentes a los caracteres en mayúsculas y en minúsculas, por ejemplo, el compilador no trataría como iguales a los identificadores que aparecen a continuación:

```
apMensaje
apmensaje
```

Otra restricción, es que no puede utilizarse como identificador a una palabra que tenga un significado predefinido dentro del lenguaje. Por lo tanto aquel que bautice a un identificador como `inline`, bien merece que su programa no sea ejecutado por errores de compilación.

En todo programa en C++, existe una función llamada `main`, que como su nombre lo dice, es la función principal y es normalmente la primera función que se ejecuta al iniciar un programa[5]. Por norma del lenguaje todo programa debe tener una y solo una función `main`. Por lo tanto, es

[3] Esto es, de la a *a* la z o de la *A* a la *Z*.

[4] El caracter de subrayado es el _.

[5] Aún cuando main es la primera función en ejecutarse, existen formas para hacer que se ejecuten otras funciones antes a través de opciones de compilación

desde main donde se regularmente se hacen los llamados a las demás funciones.

Los *Parámetros* son una lista de definiciones de variables. Una definición de variable sigue la sintaxis:

Tipo NombreVariable ;

donde:

Tipo es cualquier tipo de dato, como cualquiera de los mostrados en la Tabla 4.1.

NombreVariable es cualquier identificador para hacer referencia a la variable y que sigue las mismas reglas para un identificador expuestas anteriormente.

Ejemplos de declaraciones de variables son:

```
int  iEntero;
real rReal;
```

Al encabezado de la función, es decir, a la declaración *tipo NombreFunción (Parámetros)*, se le conoce también como prototipo de la función.

Finalmente, las *Instrucciones* son cualquier orden ejecutable del lenguaje C++. Es importante señalar que todas las *Instrucciones* deben terminar con punto y coma (;), tal y como se puede observar en la sintaxis.

```
#include <stdio.h>
#include <iostream.h>

inline void Detener ( char *apMensaje = NULL )
{
 cout << ( apMensaje ? apMensaje : "\nPresione retorno para continuar ..." );
 getchar ( );

} /*** Detener () ***/
```

Recuadro 4.1 La función Detener

Un ejemplo de función es mostrado en el Recuadro 4.1. Las primeras dos líneas, inician con un símbolo #, esto las hace instrucciones especiales que son llamadas instrucciones del preprocesador. Al compilarse un programa en C++, pasa primero a un preprocesador, el cual, identifica las instrucciones que le son dirigidas por que invariablemente inician con #, entonces las ejecuta y al terminar, se tiene un código preprocesado que se envía al compilador. La instrucción #include sirve para incluir código de librerías y tiene como principal sintaxis:

#include < *NombreLibreria* >

donde:
NombreLibrería es cualquier nombre de archivo

Para el caso de el Recuadro 4.1 se incluyen dos archivos de librerías que son estándar para cualquier compilador C++, stdio.h así como iostream.h en la primera y segunda instrucción #include respectivamente. La librería stdio.h contiene la definición de la función getchar, mientras que iostream.h contiene la definición de cout, ambos utilizados en nuestro primer ejemplo de función.

Volviendo al código del Recuadro 4.1, el indicador inline en el tercer renglón nos dice que se trata de una función en línea. A continuación está el tipo retornado, que en este caso es void, lo que significa que la función no retornará valor alguno. En seguida tenemos el nombre de la función que es Detener y entre los paréntesis ( ), un parámetro llamado apMensaje cuyo tipo es char*. El * antes de cualquier variable, hace que dicha variable sea un apuntador. Un apuntador es simplemente una variable cuyo contenido es la dirección en memoria de otra variable. Una variable char*, se utiliza regularmente para hacer referencia a una cadena caracteres, tal y como es el caso de la función Detener. Una cadena de caracteres, es una secuencia de cualquier caracter y para especificarse dentro de un programa se encierra entre comillas dobles. Ejemplos de secuencias de caracteres son:

```
"Cadena de caracteres"
"Presione retorno para continuar..."
```

A la derecha de apMensaje se puede ver un =NULL, esto significa que en caso de que no se envíe una cadena de caracteres a la función Detener, entonces a apMensaje le será asignado el valor de NULL por omisión. NULL se usa regularmente para indicar un valor nulo y normalmente significa el valor cero. Las instrucciones ejecutables encerradas entre las llaves { },

son primero cout que sirve para escribir en la salida estándar[6]. La sintaxis de cout es:

cout << *elemento1* << *elementoN* ;

donde:

elemento es cualquier variable o constante que se requiera escribir a la salida estándar.

Para explicar el funcionamiento de cout en la función Detener, es requerido el primero indicar qué es lo que hace el operador ?:. El operador ?: tiene la sintaxis:

ExpresiónAritmética ? *ValorCierto* : *ValorFalso*

donde:

ExpresiónAritmética es cualquier expresión que represente un valor numérico[7].

ValorCierto y *ValorFalso* son cualquier variable o constante.

El operador ?: funciona como una condicional que evalúa la *ExpresiónAritmética*, si dicha expresión es diferente de cero retorna el *ValorCierto*, de otra forma retorna el *ValorFalso*. En el caso específico de la función Detener, el valor de apMensaje es la *ExpresiónAritmética* que es evaluada, si su valor es diferente de NULL, entonces retorna la cadena

[6] Que regularmente es la consola

[7] tal como una suma, resta, multiplicación, división, una variable o constante numérica, en fin todo de lo que se pueda obtener un valor numérico.

apMensaje, de otra forma retorna la cadena "\nPresione retorno para continuar...". Pero, ¿Quién recibe el valor retornado por el operador? Es precisamente la función cout, que escribe la cadena de caracteres retornada a la salida estándar.

La instrucción que sigue a cout en la función Detener es el llamado a una función. La sintaxis para mandar ejecutar una función es simplemente:

NombreFunción (*Parámetros*);

donde:

NombreFunción	Es el identificador con el que se nombró a la función
Parámetros	Son los argumentos declarados como necesarios para la función.

En el caso de Detener, se hace un llamado a la función llamada getchar que no tiene parámetros, por lo que los paréntesis que están a continuación están vacíos. Lo que hace getchar, es leer un caracter de la entrada estándar que debe ser terminado por la tecla de retorno.

A la derecha de la llave } que marca la terminación de la función Detener se encuentra un letrero que dice /*** Detener () ***/, este es un comentario. Los comentarios no significan instrucción ejecutable alguna en el lenguaje, ni son necesarios para el correcto funcionamiento del programa, sino que son señalamientos que sirven de guía para el programador para reconocer más rápidamente el comportamiento de una función, o bien para facilitar la codificación de la misma. Un comentario puede hacerse de dos formas. La primera es poniendo // y todo lo que

aparezca a su derecha hasta el fin de esa línea será tomado como comentario. La segunda y más antigua es usando el /* para indicar el inicio de un comentario y el */ para indicar su finalización.

Bueno, pero ¿Qué es lo que hace la función Detener?. Recibe opcionalmente como parámetro una cadena de caracteres. Escribe la cadena de caracteres a la salida estándar a manera de mensaje, en caso de que no se envíe una cadena de caracteres, el mensaje por omisión será "Presione retorno para continuar ...". Después detiene el programa esperando a que se introduzca un retorno para continuar con la ejecución del programa.

4.2. LEER

Observe el Recuadro 4.2. Tal y como se habrá ya dado cuenta, la instrucción contenida en este recuadro es una instrucción al preprocesador, pues inicia con #. Se trata de la instrucción #define, que como dice su nombre, sirve para hacer definiciones de patrones, su sintaxis general es:

#define *PatrónDefinido DefiniciónDelPatrón*

donde:
PatrónDefinido es el identificador cuyo significado se definirá.
DefiniciónDelPatrón es el significado que tendrá el *PatrónDefinido*

```
#define Leer(x) cout << #x" = "; cin >> x
```

Recuadro 4.2 La macro Leer

En realidad, lo que hace el preprocesador con la instrucción #define, es un reemplazo de todas las ocurrencias en el programa del *PatrónDefinido* por la cadena *DefiniciónDelPatrón*. El *PatrónDefinido* puede incluir al final variables entre paréntesis () que le son mandados a manera de parámetros, en C++ a este tipo especial de definición se le llama una macro. En el Recuadro 4.2 el *PatrónDefinido* es Leer(x), se trata de una macro ya que acepta una variable llamada x que está encerrada entre los paréntesis. Si se quisiera definir más de una variable a una macro, entonces las variables se separarían por una coma (,). La *DefiniciónDelPatrón* para la macro Leer, es cout << #x" = "; cin >> x. En este caso cout[8] envía a la salida estándar a #x" = ". Una secuencia de caracteres en C++ se expresa encerrándose entre comillas dobles, por lo que " = ", es una cadena de caracteres que contiene un espacio en blanco, un signo igual y un espacio en blanco. El caracter # dentro de la *DefiniciónDelPatron* es un operador del preprocesador que retorna una cadena de caracteres con el nombre de la variable que le sigue, y como el preprocesador de C++ une dos cadenas de caracteres consecutivas, se tiene por resultado que el cout de la macro Leer escribirá a la salida estándar el nombre de la variable enviada como parámetro seguido del signo =. Después del ; que da fin a la instrucción

[8]observe que la instrucción cout termina con el punto y coma (;)

cout, sigue la instrucción cin que sirve para leer de la entrada estándar una variable, su sintaxis es similar a la de cout:

cin >> *variable* ... >> *variable* ;

donde:

variable es un variable del programa cuyo valor es leído de la entrada estándar.

Así que lo que hace la macro Leer, es escribir el nombre de la variable enviada como parámetro a la entrada estándar seguido de un signo de igualdad, esto a manera de petición para posteriormente leer el valor de la variable. Note que la definición de la macro en el Recuadro 4.2 no termina con un punto y coma (;) , si se le hubiese puesto el punto y coma entonces se podría invocar la macro en un programa como:

```
Leer ( iEntero )
```

Sin embargo se ha omitido el punto y coma (;) con el fin de darle la apariencia de una función, pues de esta forma la invocar a la macro se le tendrá que finalizar con un punto y coma:

```
Leer ( iEntero );
```

4.3. EJERCICIOS

4.3.1 *Obligatorio*

Escriba la función Detener y la macro Leer, guardándolas en un archivo llamado DepUtil.h

4.3.2 *Opcional*

Escriba el programa del Recuadro 4.3 guardándolo en un archivo llamado Detener.cpp, ejecútelo y describa su funcionamiento.

```
#include <DepUtil.h>

main ( )
{
  char *apMensaje  = "Ejecución detenida, introduzca retorno si quiere continuar";

  Detener ();
  Detener ( apMensaje );
  Detener ( "Ultima interrupción del programa, teclee se retorno para terminar" );

} /*** main ( ) ***/
```

Recuadro 4.3 El programa Detener.cpp

4.3.3 *Opcional*

Experimentemos con la función Detener, cambie la cadena "\n Presione retorno para continuar" por "Presione\n retorno para\n continuar", ejecute nuevamente Detener.cpp y describa el significado de \n para el lenguaje C++

4.3.4 *Opcional*

Pruebe la macro Leer, con el programa Leer.cpp que se muestra en el Recuadro 4.4.

```
#include <DepUtil.h>

main ( )
{
  char  cCaracter;
  int   iEntero;
  long  lLargo;
  float fFloat;

  Leer ( cCaracter );
  Leer ( iEntero   );
  Leer ( lLargo    );
  Leer ( fFloat    );

} /*** main ( ) ***/
```

Recuadro 4.4 Programa Leer.CPP

4.3.5 *Obligatorio*

Escriba la macro llamada Escribir en el archivo DepUtil.h, que recibirá un parámetro cuyo nombre escribirá a la salida estándar, seguido de un signo = y de su valor.

4.3.6 *Opcional*

Escriba un programa parecido al de Leer.cpp para probar la macro Escribir.

5. MATUTIL.H

MatUtil.h se convertirá en la segunda librería de programación para la construcción de algoritmos genéticos. Esta utilería facilitará operaciones comunes tales como la generación de números aleatorios, funciones de permutación, cálculos probabilísticos para muestreo y conversiones entre distintas bases numéricas.

5.1. INTERCAMBIAR

Resulta muy socorrido en diversas ocasiones, el tener dos variables en un programa y querer intercambiar sus valores, en el Recuadro 5.1 puede observarse una función que facilita este trabajo. Se trata de una función en línea llamada Intercambiar y que no retorna valor alguno. Puede notarse que recibe dos parámetros, pero que el nombre de cada uno esta precedido por el caracter &, esto significa que estos parámetros son llamados por referencia.

Existen dos tipos de envío de parámetros, envío por valor y envío por referencia. El enviar un parámetro por valor a una función, significa que

```
inline void Intercambiar ( int &iVariable1, int &iVariable2 )
{
 int iVariable1Clon = iVariable1;

 iVariable1 = iVariable2;
 iVariable2 = iVariable1Clon;

}/*** Intercambiar () ***/
```

Recuadro 5.1 La función Intercambiar

si se hace un cambio a algún parámetro dicho cambio no afectará a la variable con la que fue llamada la función. Por omisión, en C++ al invocar una función, las variables usadas en los parámetros son enviadas por valor, excepto para las variables tipo apuntador y arreglo[9].

El envío de parámetros por referencia, a diferencia de por valor, hace que los cambios efectuados a los parámetros, cambien también a las variables usadas para enviar los valores a la función.

El cuerpo de la función Intercambiar declara una variable que se inicializa con el valor del primer parámetro, después al primer parámetro se le asigna el segundo parámetro y la función finaliza asignando al segundo parámetro el valor del primer parámetro mediante la variable creada en la función. De esta forma al terminar la función las variables enviadas como parámetros han intercambiado sus valores.

[9]Los arreglos serán examinados con detenimiento en la sección 5.8

Observe el uso del operador = que se utiliza para la asignación. Una descripción los operadores de asignación está en la Tabla 5.1.

Operadores de asignación				
Operador	**Nombre**	**Tipo**	**Ejemplo**	**Explicación**
=	asignación	Binario	x = z	asigna a x el valor de z
+=	suma con asignación	Binario	x += z	suma x más z y asigna la suma a x
-=	resta con asignación	Binario	x -= z	resta x menos z y asigna la resta a x
*=	multiplicación con asignación	Binario	x *= z	multiplica x por z y asigna el producto a x
/=	división con asignación	Binario	x /= z	divide x entre z y asigna el cociente a x
%=	residuo con asignación	Binario	x %= z	divide x entre z y asigna el residuo a x

Tabla 5.1 Operadores de asignación del lenguaje C++ que se utilizan en este trabajo

5.2. CALCULAR FACTORIAL

Una de las funciones más utilizadas cuando se emplean procesos estadísticos, tal y como los emplean los algoritmos genéticos, es la

función para calcular el factorial de un número. Recordemos que el factorial de un número n se define como 1 para n=0 ó n=1, y como $1 \cdot 2 \cdot \ldots n\text{-}1$ para n>1.

En el Recuadro 5.2 se puede ver una función que obtiene el factorial de un número. El nombre de la función es CalcularFactorial, que recibe como parámetro una variable de tipo unsigned y retorna un valor de tipo unsigned long. La función inicia con dos de las instrucciones más usadas en C++, la instrucción if y la instrucción return.

```
unsigned long CalcularFactorial ( unsigned uNumero )
{
 if ( uNumero <= 1 ) return 1;

 unsigned long ulFactorial = uNumero;

 while ( -- uNumero ) ulFactorial *= uNumero;

 return ulFactorial;

} /*** CalcularFactorial ***/
```

Recuadro 5.2 La función Calcular Factorial

La instrucción if sirve para condicionar la ejecución de una o más instrucciones de acuerdo a si es cumplida o no una condición. Su sintaxis es:

if (*ExpresiónAritmética*) *InstruccionesVerdadero*;
else *InstruccionesFalso* ;

donde:

ExpresiónAritmética es cualquier expresión que represente un valor numérico[10]

InstruccionesVerdadero e *InstruccionesFalso* son cualquier sentencia ejecutable

La instrucción `if` evalúa la *ExpresiónAritmética* si dicha expresión resulta diferente de cero entonces se asume un valor verdadero y se ejecutan las *InstruccionesVerdadero*. De otra forma, es decir si la *ExpresiónAritmética* es igual a cero entonces se asume un valor falso y se ejecutan las *InstruccionesFalso*. La cláusula `else` es optativa, es decir, se puede prescindir de ella si no se quiere ejecutar alguna instrucción en caso falso. Si se llegara a ejecutar más de una instrucción, tanto en el caso verdadero como en el caso falso, entonces al conjunto de instrucciones se le encierra entre llaves de inicio { y fin }.

Pueden usarse en la *ExpresiónAritmética* los operadores aritméticos mostrados en la Tabla 5.2, o los operadores de aritmética booleana que están en la Tabla 5.3, o los operadores relacionales de la Tabla 5.4. Recuerde, el cero toma el significado de falso, mientras que un valor diferente de cero es tomado como verdadero.

[10]tal como una suma, resta, multiplicación, división, una variable o constante numérica, en fin de todo lo que se pueda obtener un valor numérico

OPERADORES ARITMÉTICOS				
Operador	**Nombre**	**Tipo**	**Ejemplo**	**Explicación**
*	Multiplicación	Binario	x*y	Multiplica x por y
/	División	Binario	x/y	Divide x entre y
%	Módulo	Binario	x%y	Residuo de la división de x entre y
+	Suma	Binario	x+y	Suma x más y
-	Resta	Binario	x-y	Resta x menos y
++	Incremento	Binario	x++	Incrementa x en uno después de usarla
		Unario	++x	Incrementa x en uno antes de usarla
--	Decremento	Binario	x--	Decrementa x en uno después de usarla
		Unario	--x	Decrementa x en uno antes de usarla
-	Negación	Unario	-x	Niega el valor de x

Tabla 5.2 Operadores aritméticos del C++

La instrucción return es usada para que una función termine su ejecución y retorne un valor. Su sintaxis es:

return *ValorRetornado* ;

donde:
ValorRetornado es el valor que retornará la función

OPERADORES RELACIONALES				
Operador	**Nombre**	**Tipo**	**Ejemplo**	**Explicación**
<	Menor que	Binario	x<y	1 si x es menor que y, sino 0
<=	Menor o igual que	Binario	x<=y	1 si x es menor o igual que y, sino 0
>	Mayor que	Binario	x>y	1 si x es mayor que y, sino 0
>=	Mayor o igual que	Binario	x>=y	1 si x es mayor o igual que y, sino 0
==	Igual a	Binario	x==y	1 si x es igual a y, sino 0
!=	Diferente a	Binario	x!=y	1 si x es diferente de y, sino 0

Tabla 5.3 Operadores lógicos del C++

En realidad, la instrucción return es muy flexible, puesto que puede usarse como *ValorRetornado*, desde un valor numérico constante, hasta una expresión aritmética compleja, u otras instrucciones del C++, incluso el valor retornado por el llamado a otra función. Es importante señalar que el tipo del *ValorRetornado* debe ser el mismo que el tipo de retorno indicado en el prototipo de la función.

Una vez dadas las bases anteriores, será fácil el deducir que la primera instrucción de la función CalcularFactorial, evalúa si la variable ulNumero es menor o igual a 1, en tal caso, la función retornaría 1 como resultado. Después se define una variable llamada ulFactorial cuyo tipo es

OPERADORES BOOLEANOS				
Operador	Nombre	Tipo	Ejemplo	Explicación
!	NO lógico	Unario	!x	1 si x es cero, sino 0
&&	Y lógico	Binario	x&&z	1 si x y z son diferentes de cero, sino 0
||	O inclusivo lógico	Binario	x||z	1 si x o z son diferentes de cero, sino 0

Tabla 5.4 Operadores booleanos del C++

unsigned long y a la vez se le asigna el valor de uNumero. En la siguiente línea se ejecuta una instrucción de C++ llamada while, cuya sintaxis es:

```
while ( ExpresiónAritmética ) Instruccion(es) ;
```

donde:

ExpresiónAritmética es cualquier expresión que represente un valor numérico[11].

Instruccion(es) son cualquier sentencia ejecutable en el lenguaje C++. Si existe más de una sentencia, entonces el bloque de instrucciones se encierra entre llaves de inicio { y fin }.

[11]tal como una suma, resta, multiplicación, división, una variable o constante numérica, en fin todo de lo que se pueda obtener un valor numérico.

La instrucción while ejecuta las *Instruccion(es)* repetidamente en un ciclo mientras que la *ExpresiónAritmética* sea diferente de cero. Es importante señalar que la *ExpresiónAritmética* es evaluada antes de iniciar cada ciclo, el inicial inclusive.

En la función CalcularFactorial, al inicio de cada ciclo while primero se decrementa en uno a la variable uNumero, si después del decremento uNumero es diferente de cero entonces se multiplica por ulFactorial. Al final del ciclo se retorna el valor de ulFactorial y termina de esta forma la función CalcularFactorial.

5.3. REDONDEAR

Uno de los problemas más comunes al convertir un número real a un entero, es el de obtener la aproximación más adecuada del número convertido al eliminarse su parte decimal. Tal vez la forma de aproximación más usada en la conversión de un número real a entero es la de redondeo.

El Recuadro 5.3 muestra la función Redondear, la cual retorna el redondeo de un número llamado numero de tipo double. La primera instrucción crea una variable double llamada dNumeroParteEntero, misma que se inicializa con el resultado de la función floor con parámetro numero. La función estándar floor, definida en math.h, recibe como parámetro un número de tipo double y retorna el piso del parámetro, es decir, el número

```
inline long Redondear ( double dNumero )
{
 double dNumeroParteEntero = floor ( dNumero );
 double dNumeroParteFraccionario = dNumero - dNumeroParteEntero;

 if ( dNumeroParteFraccionario >= 0 )
  return (long)(dNumeroParteFraccionario<.5?dNumeroParteEntero:++dNumeroParteEntero);
 else
  return (long)(dNumeroParteFraccionario>=-.5?dNumeroParteEntero:--dNumeroParteEntero);

} /*** Redondear () ***/
```

Recuadro 5.3 La función Redondear

entero más grande menor o igual que el parámetro. Por ejemplo, el piso de 4.15 es 4.

Posteriormente se declara otra variable que contiene la parte fraccionaria del parámetro. Después mediante una sentencia if se pregunta si la parte fraccionaria es mayor o igual a cero, esto sirve para detectar si el número a redondear es positivo o negativo. En caso de que el número a redondear sea positivo, si la parte fraccionaria es menor que .5, entonces se retorna la parte entera, de otra forma se retorna la parte entera incrementada en uno. En caso de que el número a redondear sea negativo, si la parte fraccionaria es mayor o igual a -.5 entonces se retorna la parte entera, de otra forma se decrementa la parte entera en uno.

5.4. NÚMEROS ALEATORIOS

Un algoritmo genético inmiscuye diversos factores aleatorios en varios de sus pasos. Para ello se han creado las funciones InicializarAleatorios, y GenerarAleatorios. La primera función se muestra en la Recuadro 5.4, su objetivo es inicializar el generador de números aleatorios.

```
inline void InicializarAleatorios ( void )
{
 srand ( (unsigned) time (NULL) );

} /*** Inicializar () ***/
```

Recuadro 5.4 La función InicializarAleatorios

La única línea con la que cuenta, ejecuta la función estándar de C++ srand, que inicializa el generador de números pseudoaleatorios y que recibe como parámetro un número inicial, llamado también semilla, para la secuencia de números pseudoaleatorios. El parámetro enviado a srand, resulta ser el valor retornado por otra función estándar de C++ llamada time, si a esta función se le envía como parámetro el valor NULL, entonces retorna el número de segundos transcurridos desde la hora 00:00:00 GMT del primero de enero de 1970.

El tipo de variable retornado por time es dependiente de cada máquina en específico y el tipo requerido como parámetro por srand es un unsigned; para remediar este caso se puede utilizar una conversión de tipo, llamada

también tipificación. La tipificación consiste en colocar a la izquierda del valor, el tipo al cual se quiere convertir encerrado entre paréntesis. En InicializarAleatorios la parte (unsigned) a la izquierda de time, sirve para indicar la conversión del valor retornado por time al tipo unsigned.

La tipificación es ampliamente usada y altamente recomendada, puesto que de no hacerse, no puede asegurarse de manera alguna que las variables sean evaluadas con su verdadero valor. El no usar la tipificación puede ocasionar poca portabilidad y ejecuciones erráticas del código.

```
inline int GenerarAleatorio ( int iLimiteInferior, int iLimiteSuperior )
{
 if ( iLimiteInferior==iLimiteSuperior ) return iLimiteInferior;
 if ( iLimiteInferior>iLimiteSuperior ) Intercambiar(iLimiteInferior,iLimiteSuperior);

 double dLimiteInferior = (double)iLimiteInferior - 0.5;

 return (int) Redondear (
  (0.49+(double)iLimiteSuperior-dLimiteInferior)/(double)RAND_MAX*(double)rand()
  +dLimiteInferior
                         );

} /*** GenerarAleatorio () ***/
```

Recuadro 5.5 La función GenerarAleatorio

La función GenerarAleatorio es mostrada en el Recuadro 5.5. Esta función retorna un número aleatorio de tipo int entre dos límites numéricos, uno iLimiteSuperior y otro iLimiteInferior, que le son enviados como parámetros. Si iLimiteInferior fuera igual a iLimiteSuperior, se retornaría precisamente iLimiteSuperior. En caso de que iLimiteInferior fuera mayor que iLimiteSuperior, se intercambian los valores de estas variables.

Para obtener un número aleatorio se utiliza la función estándar rand que se encuentra definida en la librería math.h. La función rand no recibe parámetros y retorna un número pseudoaleatorio. Un número aleatorio entre dos números podría obtenerse fácilmente siguiendo la fórmula:

$$Z=\frac{S-I}{R_m}\times R+I \qquad \text{Fórmula 5.1}$$

donde:

- Z es el número aleatorio entre dos límites resultante
- S es el límite superior
- I es el límite inferior
- R es un número aleatorio
- R_m es el valor máximo que puede tener R

La última línea de la función Redondear retorna precisamente el resultado de la Fórmula 5.1. Note el uso de una constante estándar definida en math.h llamada RAND_MAX que contiene el valor máximo devuelto por la función rand. Observe también que se han incrementado los límites en .5 para evitar los efectos colaterales debidos a que se usa una función de redondeo.

5.5. PRUEBAS REPETIDAS

Nuestros algoritmos genéticos, en varios de sus procesos requieren de funciones de muestreo para pruebas con reemplazo. Reemplazo significa que la población de la que se obtuvo la muestra sigue siendo exactamente la misma después de que se ha tomado la muestra. En un algoritmo

genético es necesario el conocer cuántas muestras se deben tomar dada una cantidad determinada de elementos, si se desea elegir un elemento en específico con cierta probabilidad de selección y hay reemplazos en cada muestra. En otras palabras, ¿Cuántas pruebas tengo que realizar para obtener al elemento x con una probabilidad y, si tengo n elementos y existe reemplazo después de cada prueba? Este tipo de problemas son conocidos dentro del campo estadístico como "cálculo probabilístico de pruebas repetidas". En realidad tal vez no encontremos en un libro de estadística la fórmula exacta para responder una pregunta tan específica como la hecha en este párrafo, sin embargo, cualquier libro de estadística elemental puede darnos las bases suficientes para nosotros llegar a esta fórmula.

La probabilidad de selección de un elemento x en una sola prueba para una población de n elementos es de:

$$P(x)=\frac{1}{n}$$

donde:

$P(x)$ es la probabilidad de éxito para un evento x en una sola prueba

Siendo que la probabilidad de ocurrencia puede ir desde 0 para la seguridad de que no ocurrirá un evento, hasta 1 para la total seguridad de que ocurrirá un evento, podemos obtener que la probabilidad de fracaso para x, es decir la selección no exitosa de x en una sola prueba, para una población de n elementos es de:

$$Q(x)=1-P(x) \qquad \text{Fórmula 5.3}$$

$$Q(x)=1-\frac{1}{n}$$

$$Q(x)=\frac{n}{n}-\frac{1}{n}$$

$$Q(x)=\frac{n-1}{n} \qquad \text{Fórmula 5.6}$$

donde:

$Q(x)$ es la probabilidad de fracaso de un evento x

Sin embargo, por cada prueba repetida con reemplazo, la probabilidad de fracaso disminuye en forma exponencial por cada prueba realizada, es decir, para z pruebas repetidas con reemplazo:

$$Q_z(x)=(Q(x))^z \qquad \text{Fórmula 5.7}$$

donde:

$Q_z(x)$ es la probabilidad de fracaso para un evento x después de z pruebas repetidas con reemplazo.

De esta manera, de la Fórmula 5.3 y de la Fórmula 5.7 podemos obtener la probabilidad de éxito para un evento x en z pruebas repetidas con reemplazo, restando 1 a $Q(x)$ de la siguiente forma:

$$P_z(x)=1-Q_z(x) \qquad \text{Fórmula 5.8}$$

donde:

$P_z(x)$ es la probabilidad de éxito para un evento x después de z pruebas repetidas con reemplazo.

Desarrollando la Fórmula 5.7 en la Fórmula 5.8:

$$P_z(x)=1-(Q(x))^z \qquad \text{Fórmula 5.9}$$

Desarrollando la Fórmula 5.6 en la Fórmula 5.9:

$$P_z(x)=1-(\frac{n-1}{n})^x \qquad \text{Fórmula 5.10}$$

Despejando z en la Fórmula 5.10:

$$P_z(x)-1=-(\frac{n-1}{n})^x$$

$$1-P_z(x)=(\frac{n-1}{n})^x$$

$$\log(1-P_z(x))=\log(\frac{n-1}{n})^x$$

$$\log(1-P_z(x))=z\log(\frac{n-1}{n})$$

$$z=\frac{\log(1-P_z(x))}{\log(\frac{n-1}{n})} \qquad \text{Fórmula 5.15}$$

En la Fórmula 5.15 tenemos despejada z, esto quiere decir que podemos obtener con esta fórmula el número de pruebas con reemplazo (z) que son necesarias para seleccionar un elemento (x), dada una probabilidad de selección para dicho elemento ($P_z(x)$) y el número total de elementos (n).

La traducción de la Fórmula 5.15 al lenguaje C++, es como se muestra en el Recuadro 5.6, donde se puede observar que la función que realiza la tarea es llamada `CalcularPruebasRepetidas`, que es una función en línea y recibe dos parámetros, el primero nombrado `ulEventosCantidad` es la cantidad de eventos posibles, es decir, la variable n en la Fórmula 5.15.

El segundo, llamado fSeleccionProbabilidad, es la probabilidad de seleccionar a un elemento en específico, esto es, la variable $P_s(x)$ en la Fórmula 5.15.

```
#include <math.h>

inline unsigned long CalcularPruebasRepetidas (
                        unsigned long ulEventosCantidad, float fSeleccionProbabilidad
                                                )
{
 if ( fSeleccionProbabilidad <= 0.0 ) return 0;
 if ( ulEventosCantidad <= 1 ) return ulEventosCantidad;
 if ( fSeleccionProbabilidad >= 1 ) fSeleccionProbabilidad = .99;

 return (unsigned long) Redondear (
  log10(1-(double)fSeleccionProbabilidad)
  /log10((double)(ulEventosCantidad-1)/(double)ulEventosCantidad)
                                  );

} /*** CalcularPruebasRepetidas () ***/
```

Recuadro 5.6 La función CalcularPruebasRepetidas

Las primeras tres sentencias con las que inicia CalcularPruebasRepetidas, son sentencias if que sirven de condicionales para validación. Si la probabilidad de seleccionar un evento en específico (fSeleccionProbabilidad) es igual a cero, entonces, obviamente el número de pruebas que deben realizarse son cero. Si la cantidad de eventos posibles (ulEventosCantidad) es menor o igual a uno, entonces se retorna esta misma cantidad, puesto que la cantidad de eventos posibles debe ser mayor a 1 para poder tener un factor probabilístico y emplear la Fórmula 5.15. Finalmente si la probabilidad de seleccionar un evento específico (fSeleccionProbabilidad)

es mayor o igual a uno, entonces, dicha probabilidad es reasignada a .99 para mantenerla en ámbitos probabilísticos.

La última instrucción de CalcularPruebasRepetidas es con la que se calcula la Fórmula 5.15. Observe los dos llamados a la función estándar log10(), que está definida en la librería estándar math.h, y que retorna el logaritmo en base 10 de un número double enviado como parámetro, por ello al invocar a log10 dentro de CalcularPruebasRepetidas, se hace una tipificación a double. El resultado finalmente retornado es redondeado con la función Redondear, tipificando el resultado a un unsigned long.

5.6. OBTENER UN DÍGITO

En ciertos procesos involucrados con los algoritmos genéticos, es requerido el obtener un dígito en particular de un número si se sabe de cuántos dígitos está compuesto dicho número. Por ejemplo, ¿Cómo obtener el tercer dígito del número 8736737 si se sabe que tiene 7 dígitos? Para responder a esta pregunta he propuesto la siguiente fórmula:

$$d = \text{trunc}\left(\frac{n \% 10^{l-r+1}}{10^{l-r}}\right) \qquad \text{Fórmula 5.16}$$

para todo:

$0 < r \leq l$

$n > 0$

donde:

d es el dígito buscado

n	es el número del cuál se extraerá el dígito
l	es la longitud de *n*, es decir, el número de dígitos por los que está compuesto *n*
r	es la posición del dígito que se quiere extraer de *n*
%	significa el residuo de una división, tal cual es su significado dentro de C++
trunc()	es una función que retorna la parte entera de un número real.

Hagamos un análisis de la Fórmula 5.16. El residuo $n\%10^{l-r+1}$, sirve para obtener al número *n* sin sus primeros $r-1$ dígitos. El divisor, 10^{l-r}, sirve para obtener en el cociente final el dígito buscado *d*, pues se eliminarán los *l-r* últimos dígitos de *n*. Para una mayor comprensión, hagamos un ejemplo paso a paso. Para obtener el tercer dígito del número 8736737 si se sabe que tiene 7 dígitos, substituyamos las variables:

$n = 8736737$

$l = 7$

$r = 3$

entonces:

$10^{l-r+1}=10^{7-3+1}=100000$

y:

$n\%10^{l-r+1}=8736737\%100000=3673$

con lo que se eliminan los primeros $r-1$ dígitos de *n*, es decir se eliminan el 8 y el 7. Continuando, tenemos que:

$10^{l-r}=10^{7-3}=10000$

sustituyendo todo:

$$d = \text{trunc}(\frac{n\%10^{l-r+1}}{10^{l-r}}) = \text{trunc}(\frac{3673}{10000}) = \text{trunc}(3.673) = 3$$

con lo que se consiguió nuestro objetivo, obtener el tercer dígito de 8736737, a sabiendas de que contiene siete dígitos.

```
inline int ObtenerDigito (
          unsigned long ulNumero, unsigned uNumeroLong, unsigned uDigitoRequerido
                    )
{
 return (int) (
     (ulNumero%(unsigned long)pow((double)10,(double)(uNumeroLong-uDigitoRequerido+1)))
     / pow ( (double)10, (double)(uNumeroLong - uDigitoRequerido) )
           );

} /*** int ObtenerDigito ***/
```

Recuadro 5.7 La función ObtenerDigito

Una función en C++ que ejecuta la Fórmula 5.16 es mostrada en el Recuadro 5.7. Como puede observarse la fórmula ha sido transcrita textualmente en la función ObtenerDigito, que recibe tres parámetros, el primero es el número del cuál se extraerá el dígito, el segundo es la longitud del número y finalmente, el tercero indica la posición del dígito requerido. La función está en línea y su código retorna el dígito deseado en un formato int. Para elevar 10 a la potencia requerida, se utiliza la función power, que recibe dos parámetros de tipo double, el primero es el número base y el segundo es el exponente. La función power, está definida en la librería estándar math.h.

5.7. LA FUNCIÓN PERMUTAR

Las permutaciones de los dígitos de un número son el resultado del arreglo de todos los dígitos del número en un orden definido. Por ejemplo, las permutaciones de los dígitos para el número 123 serían los mostrados en la Tabla 5.5, en ella llamemos a 123 la primera permutación, 132 la segunda, y así sucesivamente hasta llegar a la 321, que es la sexta permutación. Supóngase que nos encontramos frente al problema de buscar la *n*ésima permutación de los dígitos de un número cualquiera, esto es, que deseamos responder a preguntas del tipo, ¿Cuál es la *n*ésima permutación de los dígitos del número *z*?

123
132
213
231
312
321

Tabla 5.5 Permutaciones de los dígitos del número 123.

En adelante utilizaremos los símbolos $z_n!$ para denotar la *n*ésima permutación de los dígitos del número z. Para poder llegar a una solución a $z_n!$ he propuesto el método que se describe en los siguientes párrafos.

Comencemos por observar de manera deductiva las permutaciones más sencillas, iniciando con las permutaciones de un número de dos dígitos, tal y como 58:

58

85

como es lógico, este número consta únicamente de dos permutaciones. Lo que es en realidad importante de esta permutación tan primitiva, es que se pueden deducir las dos primeras soluciones generales para el cálculo de $z_n!$:

1ª solución general

La primera permutación de un número z, es precisamente el número z, esto es: $z_1! = z$

2ª solución general

La segunda permutación de z para $z<100$, es igual al simple intercambio entre los dos dígitos de z, lo que se puede realizar con la fórmula:

$$z_2! = ((z\%10)\times 10)+\text{trunc}(\frac{z}{10})$$

donde:

z es el número del cuál se busca la segunda permutación de sus dígitos

trunc() es una función que retorna la parte entera de un número real

% es el residuo de la división, tal y como es su significado dentro del lenguaje C++.

Por ejemplo, el cálculo de $85_2!$ se resolvería de la siguiente forma:

$$85_2! = ((85\%10)\times10)+\text{trunc}(\frac{85}{10}) = (5\times10)+8 = 58$$

Observemos una vez más la permutación para un número de tres dígitos mostrada en la Tabla 5.5, es claro que puede dividirse el listado de permutaciones en la forma en que cambia el número de más a la izquierda. A cada uno de estos cambios del dígito de más a la izquierda, le llamaremos región denotándolo con la letra *r*, de esta forma tenemos tres regiones, que están indicadas en la Tabla 5.6.

Obsérvese que en la Tabla 5.6 cada dos permutaciones se cambia de región. En adelante, llamaremos factor de cambio, al número de permutaciones requeridas para cambiar de una región a otra y lo denotaremos con la letra *f*. Es claro que:

$$f = (L-1)!$$

donde:

L es la longitud, es decir, el número de dígitos por los que está compuesto el número que se está permutando.

f es el factor de cambio para un número.

1ª REGIÓN
123
132
2ª REGIÓN
213
231
3ª REGIÓN
312
321

Tabla 5.6 Regiones de una permutación ordenada ascendente de un número con tres dígitos

Por ejemplo, para el número 123, cuya longitud es de 3, el factor de cambio sería igual a:

$$f = (3-1)! = 2$$

Con estas bases podemos determinar el número de región en la que se encuentra una permutación buscada. Por ejemplo, ¿En qué región se localiza $123_4!$ (léase: la cuarta permutación de 123)? La respuesta se puede obtener con la fórmula:

$$r=trunc(\frac{n-1}{f})+1 \qquad \text{Fórmula 5.27}$$

donde:

r es el número de región de $z_n!$

n es el número de permutación buscada de un número z

De esta manera, podemos obtener el número de región donde se localiza $123_4!$, sustituyendo en la Fórmula 5.27 tenemos que:

$$r = trunc(\frac{4-1}{2})+1 = 1+1 = 2$$

así, ahora ya sabemos que $123_4!$ se encuentra en la región dos, resultado que se puede comprobar con la lista de permutaciones ordenadas proporcionadas en la Tabla 5.6.

Pero note algo importante, el número de región para $z_n!$, nos da también la posición del dígito con que inicia $z_n!$, por ejemplo la región de $123_4!$ es igual a 2, lo que significa que $123_4!$ iniciará con el segundo dígito de 123, esto es, con el dígito 2. Por lo tanto podemos encontrar el dígito con que iniciará $z_n!$, con la siguiente fórmula:

$$i = (10^{L-1} \times S(z,r)) \qquad \text{Fórmula 5.29}$$

donde:

i es el dígito con el que iniciará $z_n!$

L es la longitud de z, es decir, el número de dígitos que tiene z

z es un número del cuál se obtendrá la $n$ésima permutación

r es el número de región de $z_n!$

S es una función que obtiene el dígito de la posición r, del número z

Por ejemplo, para $123_4!$, podemos obtener el primer dígito con que inicia $z_n!$, despejando en la Fórmula 5.29 de la forma siguiente:

$$i = (10^{3-1} \times S(123,2)) = 100 \times 2 = 200$$

En realidad, esta fórmula no ha arrojado el segundo dígito de 123 con el que iniciaría $123_4!$, puesto que bastaría la parte $S(z,r)$ de la Fórmula 5.29 para obtener el dígito 2, sino que hemos introducido el truco de multiplicar $S(z,r)$ por 10^{L-1}, para de esta forma, llegar a un número tal que solo sea necesario sumarle los dígitos restantes de la permutación final buscada. Por ello es que el ejemplo anterior nos da como resultado 200.

Ahora bien, analicemos otra vez la Tabla 5.6. Observe que podemos subdividir el problema de buscar $z_n!$, en primero obtener el dígito más a la izquierda de $z_n!$ y después buscar otra permutación de z que complete los dígitos restantes. Esto lo podemos enunciar en una tercera solución general:

3ª solución general

para $z \geq 100$:

$$z_n! = (10^{L-1} \times S(z,r)) + (E(z,r)_{n-((r-1)\times f)}!) \qquad \text{Fórmula 5.31}$$

donde:

- z es el número del cual se obtendrá la *n*ésima permutación
- n es el número de permutación buscada para el número z.
- L es la longitud del número z, esto es, el número de dígitos que contiene z
- r es el número de región en el cual se encuentra $z_n!$
- f es el factor de cambio de región para $z_n!$
- S es una función que obtiene el dígito r del número z
- E es una función que elimina el dígito r del número z

En realidad se trata de la Fórmula 5.29, a la que se ha sumado al final la búsqueda de una permutación de los dígitos restantes del número *z*. El número de permutación que se suma al final es precisamente *n* disminuido en *r-1* multiplicado por *f*, esto es para compensar la eliminación de un dígito de *z* con la función *E*. De esta forma en la búsqueda de la siguiente permutación, se asegura que se obtendrá el número de permutación correcto. Por ejemplo, para $123_4!$ sustituyendo en la Fórmula 5.31 tenemos:

$$123_4! = (10^{3-1} \times S(123,2)) + (E(123,2)_{4-((2-1)\times 2)}!)$$

$$= 200 + (13_2!)$$

y sustituyendo con la 2ª solución general tenemos que:

$$= 200 + ((13\%10)\times 10) + \text{trunc}(\frac{13}{10})$$

$$= 200 + 31 = 231$$

con esto, el resultado final es la permutación deseada, es decir $123_4! = 321$.

Una función cuyo propósito es buscar la *n*ésima permutación de los dígitos de un número es mostrada en el Recuadro 5.8. Esta función es llamada Permutar, recibe tres parámetros, el primero es el número que se permutará, el segundo es la longitud del número y el tercero es la permutación requerida; así mismo la función retornará la permutación requerida en un formato unsigned long. Las dos primeras sentencias if son tanto la 1ª como la 2ª soluciones generales a la permutación de los dígitos de un número, que fueron propuestas anteriormente y que se presentan

traducidas al lenguaje C++. En el caso de que se pudieran aplicar estas soluciones, la función terminaría retornando el valor correspondiente. En el caso de no aplicarse la 1ª o la 2ª soluciones, se calcularían los valores del factor de cambio y de región para finalmente ejecutar una instrucción return que terminaría la función devolviendo el valor correspondiente a la permutación pedida. Note que dentro de esta última instrucción return, la función Permutar se hace un llamado a sí misma con otros parámetros, de tal forma que se cumpla la 3ª solución general. Cuando una función se invoca a sí misma dentro de su código, se dice que se trata de una función recursiva, dado que se define en términos de sí misma.

```
unsigned long Permutar (
   unsigned long ulNumero, unsigned uNumeroLong, unsigned long ulPermutacionRequerida
                          )
{
 if ( ulPermutacionRequerida == 1 ) return ulNumero;
 if (ulNumero<100)
   return (unsigned long)( (ulNumero % 10) * 10) + (unsigned long)(ulNumero / 10 );

 unsigned long ulFactorDeCambio = CalcularFactorial ( uNumeroLong - 1 );
 unsigned uRegion = (unsigned)( (ulPermutacionRequerida-1) / ulFactorDeCambio ) + 1;

 return (
           (unsigned long)pow((double)10,(double)(uNumeroLong-1))
         * ObtenerDigito(ulNumero, uNumeroLong, uRegion )
       )
        + Permutar (
                      EliminarDigito ( ulNumero, uNumeroLong, uRegion),
                      uNumeroLong - 1,
                      ulPermutacionRequerida - ((uRegion - 1)
                      * ulFactorDeCambio)
                    );

} /*** Premutacion ***/
```

Recuadro 5.8 La función Permutar

Observe que en Permutar se hace uso de la función EliminarDigito que se a optado por dejar como un ejercicio obligatorio, una descripción completa de esta función puede encontrarse al final de este capítulo en la sección de ejercicios.

5.8. CONVERSIÓN DECIMAL A BINARIA

Dentro del lenguaje C++, un arreglo no es otra cosa que una secuencia de elementos del mismo tipo, cada uno de los cuales puede ser referenciado a través de un índice. Para declarar un arreglo en lenguaje C++, se sigue la sintaxis:

Tipo NombreVariable [*Tamaño*];

donde:

Tipo	es cualquier tipo válido de variable
NombreVariable	es un identificador para la variable.
Tamaño	es el número de elementos que contendrá el arreglo.

Ejemplos de arreglos son los mostrados en el Recuadro 5.9, donde en la primera línea aparece la declaración de un arreglo llamado iVectorEnteros que consta de 30 elementos de tipo int, la segunda línea contiene una variable llamada cCadena que contiene 20 elementos de tipo char, y finalmente, en la última línea aparece un arreglo llamado fVector que esta formado por 2 elementos de tipo float.

```
int iVectorEnteros [ 30 ];
char cCadena [ 20 ];
float fVector [ 2 ];
```

Recuadro 5.9 Ejemplos de declaraciones de arreglos

El arreglo tiene la facilidad de poder accesarse mediante un índice, con la peculiaridad de que los elementos se enumeran comenzando siempre desde el cero, por lo que los índices regularmente estarán decrementados a uno con respecto al verdadero número de elemento al que queremos hacer referencia. Ejemplos pueden observarse en el Recuadro 5.10, donde en la primera línea se le asigna al catorceavo elemento de iVectorEnteros el número 4, y en la segunda línea se le asigna al primer elemento de fVector el número 3.1416.

```
iVectorEnteros [15] = 4;
fVector        [ 0] = 3.1416;
```

Recuadro 5.10 Ejemplos de acceso a un arreglo

Dado que los algoritmos genéticos que aquí desarrollaremos, trabajan mediante el sistema numérico binario, es bastante deseable el tener una función que pueda convertir un número decimal a su equivalente binario. La conversión de un número decimal a un número binario se puede lograr dividiendo el número decimal entre dos, guardando el residuo y volviendo a dividir el cociente hasta que este sea cero. Una función que realiza la conversión decimal-binario es mostrada en el Recuadro 5.11, esta función es llamada ConvertirDecBinArr y su objetivo es convertir un número decimal

a su equivalente en base binaria, depositando cada dígito binario en formato caracter dentro de un arreglo de tipo caracter.

```
#include <stdio.h>

char *ConvertirDecBinArr (
                unsigned long ulNumero, char *apBinario, unsigned long ulLongBinario
                        )
{

 while ( ulLongBinario && (ulNumero != 0) )
 {
  apBinario [ --ulLongBinario ] = (ulNumero % 2) == 1 ? '1' : '0';
  ulNumero = ulNumero / 2;

 } /*** while ( ulLongBinario && (ulNumero != 0) ) ***/

 while ( ulLongBinario ) apBinario [ --ulLongBinario ] = '0';

 return apBinario;


} /*** ConvertirDecBinArr () ***/
```

Recuadro 5.11 La función ConvertirDecBinArr

Como puede observarse en el Recuadro 5.11 la función recibe como primer parámetro el número en base 10 que será convertido en base 2, el segundo parámetro es un apuntador a una variable de tipo caracter y el tercer parámetro es el número de elementos que contiene el apuntador.

Note en la función ConvertirDecBinArr algo muy peculiar dentro de su primer ciclo while, y es que en la primera instrucción de este ciclo, a la variable apBinario le siguen corchetes [] y dentro de ellos un número a manera del índice usado en una variable tipo arreglo. Esto se debe a que

los apuntadores pueden ser utilizados de igual manera que los arreglos. La similitud en su uso es porque los elementos que conforman a un arreglo son almacenados en la memoria del computador en forma continua, tal y como se guardan los elementos a los que apunta un apuntador. La instrucción `while` que forma el cuerpo de la función se ejecutará mientras la variable `ulLongBinario`, usada para accesar los elementos de la variable apuntador, sea igual a cero, ó que el número decimal sea igual a cero lo que indicaría que la conversión a terminado. En cada ciclo se calcula un nuevo dígito binario y se reinicializa la variable que contiene el número decimal como preparación para un nuevo ciclo.

Antes de finalizar la función se utiliza un segundo ciclo `while` con la intención de que si la longitud del arreglo donde se deposita el número binario resulta sobrada, entonces a las localidades más significativas, se les asignen dígitos ceros, de tal manera que no se afecte el resultado y se inicialice todo el arreglo. Por último la función termina retornando el apuntador al arreglo donde se almacenó el número binario.

Observe en esta función, que un solo caracter en C++ es delimitado por comillas simples (''), a diferencia de las cadenas de caracteres que son delimitadas con comillas dobles ("").

5.9. EJERCICIOS

5.9.1 *Obligatorio*

Introduzca todas las funciones desarrolladas en este capítulo dentro del archivo `MatUtil.h`, teniendo cuidado de hacer las inclusiones a los archivos de librería descritos para las funciones estándar que se han utilizado: `stdlib.h`, `math.h`, y `time.h`.

5.9.2 *Obligatorio*

Escriba la función `Intercambiar` que recibirá dos parámetros tipo `float` e intercambiará sus valores.

Introduzca esta función en el archivo `MatUtil.h`.

5.9.3 *Opcional*

Introduzca en un archivo llamado `CalcularFactorial` el programa mostrado en el Recuadro 5.12. Ejecútelo y describa su funcionamiento.

```
#include <DepUtil.h>
#include <MatUtil.h>


main ()
{
 int iContador = 16;
 while (  iContador-- ) Escribir ( CalcularFactorial ( iContador ) );
} /*** main () ***/
```

Recuadro 5.12 El programa factorial.cpp

5.9.4 *Opcional*

En la función CalcularFactorial cambie la línea while (--uNumero) ulFactorial*=uNumero por la línea while (uNumero--) ulFactorial*=uNumero. Una vez hecho el cambio ejecute otra vez el programa factorial.cpp, ¿Cuál es el resultado que reporta esta función y porqué?.

5.9.5 *Opcional*

Introduzca el programa del Recuadro 5.13 archivándolo como redondeo.cpp. Ejecútelo y describa su funcionamiento.

5.9.6 *Opcional*

Introduzca el programa del Recuadro 5.14 archivándolo como aleatorios.cpp. Ejecútelo y describa su funcionamiento.

5.9.7 *Obligatorio*

Escriba la función RealizarEvento, esta función recibirá un parámetro float que representará la probabilidad de ocurrencia de un evento. RealizarEvento retornará 1 como tipo int, si el evento es realizable ó 0 si el evento no es

```
#include <MatUtil.h>
#include <DepUtil.h>

main ( )
{

 Escribir ( Redondear ( .5 ));
 Escribir ( Redondear ( -.5 ));
 Escribir ( Redondear ( 1.5 ));
 Escribir ( Redondear ( -1.5 ));
 Escribir ( Redondear ( 1.6 ));
 Escribir ( Redondear ( -1.6 ));

}
```

Recuadro 5.13 El programa redondeo.cpp

```
#include <DepUtil.h>
#include <MatUtil.h>

main ()
{

 InicializarAleatorios ();

 int iContador = 20;
 while ( iContador --) Escribir ( GenerarAleatorio ( -1, 1 ) );

 Detener ();

 iContador = 20;
 while ( iContador --) Escribir ( GenerarAleatorio ( (float)-21.5, 21.5) );

} /*** main () ***/
```

Recuadro 5.14 El programa aleatorios.cpp

realizable. Para determinar si el evento es realizable o no, se compara la probabilidad de ocurrencia del evento contra un valor aleatorio, teniendo

en cuenta que el valor aleatorio máximo generado por rand está definido por la constante RAND_MAX. Por otra parte, la función debe validar que la probabilidad de ocurrencia del evento sea mayor que cero y menor que uno, ya que si se rebasan esos valores dejaría de ser un evento probabilístico para convertirse en uno determinístico

5.9.8 *Opcional*

Ejecute el programa nombrado pruebas.cpp que mostrado en el Recuadro 5.15 y describa su funcionamiento.

5.9.9 *Opcional*

Ejecute el programa del Recuadro 5.16 guardándolo bajo el nombre de obtenerdigito.cpp, describa su funcionamiento.

```
#Include <DepUtil.h>
#Include <MatUtil.h>

main ()
{
 Escribir ( CalcularPruebasRepetidas (   1, 0) );
 Escribir ( CalcularPruebasRepetidas (   0, 1) );
 Escribir ( CalcularPruebasRepetidas (   1, 1) );
 Escribir ( CalcularPruebasRepetidas ( 100, 1) );
 Escribir ( CalcularPruebasRepetidas ( 100, 0) );
 Escribir ( CalcularPruebasRepetidas ( 100,.5) );
 Escribir ( CalcularPruebasRepetidas ( 100,.2) );
 Escribir ( CalcularPruebasRepetidas ( 100,.8) );

} /*** main () ***/
```

Recuadro 5.15 El programa pruebas.cpp

```
#include <MatUtil.h>
#include <DepUtil.h>

main ()
{
  unsigned       uContador    = 8;
  unsigned       uNumeroLong = 7;
  unsigned long ulNumero     = 1234567;

  while (-- iContador)
    Escribir ( ObtenerDigito ( ulNumero, uNumeroLong, uContador ) );

} /*** main () ***/
```

Recuadro 5.16 El programa obtenerdigito.cpp

5.9.10 *Obligatorio*

Escriba una función llamada EliminarDigito, cuyo objetivo sea inverso al de ObtenerDigito, esto es, que elimine el dígito de un número. A esta función se le enviarán tres parámetros, el primero un número, el segundo la longitud del número y el tercero el número de dígito que será eliminado. Para ejemplificar el funcionamiento de EliminarDigito, supongamos que se le envía como número 8654320 en el primer parámetro, la longitud de 7 en el segundo parámetro y el 5 como lugar del dígito a eliminar en el tercer parámetro, entonces la función retornaría el número 865420.

Incluya la función ObtenerDigito en el archivo MatUtil.h.

5.9.11 *Opcional*

Escriba un programa que pruebe el funcionamiento de la función EliminarDigito.

5.9.12 *Opcional*

Introduzca el programa del Recuadro 5.17 en un archivo llamado permutar.cpp, ejecútelo y describa su funcionamiento.

```
#include <DepUtil.h>
#include <MatUtil.h>

main ()
{
 unsigned long ulNumero = 1234567;
 unsigned uTotalDigitos = 7;
 unsigned uContador = 5041;
 while(--uContador)Escribir(uContador<<":"<<Permutar(ulNumero,uTotalDigitos,uContador));
} /*** main () ***/
```

Recuadro 5.17 El programa permutar.cpp

5.9.13 *Obligatorio*

Para convertir un número binario a sistema decimal, se suma cada dígito del número binario multiplicado por 2 elevado a la potencia que indique su posición de derecha a izquierda iniciando desde cero. Por ejemplo, el número binario 110 se traduciría a decimal de la forma:

$$1\times2^2+1\times2^1+0\times2^0 = 6$$

con lo que tenemos que 110_2 es igual a 6_{10}. Programe usted mismo la función ConvertirBinArrDec cuyo objetivo será conviertir un número decimal a binario. La función aceptará dos parámetros, el primero será un apuntador de tipo char que contendrá los dígitos de un número binario en formato caracter, el segundo será la longitud del primer parámetro. La función además retornará en formato unsigned long el número convertido en base diez. Recuerde que en caso de que el arreglo tenga más localidades de las necesarias para almacenar el número binario, entonces

las posiciones más significativas tendrán ceros. Por ejemplo, si el arreglo es de cinco localidades y tiene almacenado el número tres binario, entonces el número se representaría como "00011".
Inserte la función ConvertirBinArrDec a el archivo MathUtil.h

5.9.14 *Opcional*

Introduzca, ejecute y describa el funcionamiento del programa mostrado en el Recuadro 5.18, almacenándolo como conversión.cpp.

```
#include <MatUtil.h>
#include <DepUtil.h>

main ()
{
 unsigned long ulLongBinario = 9,
               ulNumero      = 21;
 char            aBinario [10];

 aBinario [9] = NULL;

 while ( ulNumero-- )
 {
  Escribir ( ConvertirDecBinArr ( ulNumero, aBinario, ulLongBinario ) );
  Escribir ( ConvertirBinArrDec ( aBinario, ulLongBinario ) );
  Detener ();
 }

} /*** main () ***/
```

Recuadro 5.18 El programa conversion,cpp

6. StrUtil.H

Los algoritmos genéticos aquí expuestos, tienen su representación más básica a través de los caracteres, es decir de secuencias de variables tipo char, mejor conocidas como *strings*. En esta sección, se conocerán los conceptos básicos para el manejo de secuencias de caracteres, al mismo tiempo se desarrollará una sencilla pero valiosa utilería llamada StrUtil.h, cuyo propósito es el manejo de secuencias de caracteres y que será de gran ayuda para la construcción de algoritmos genéticos.

6.1. Cadenas de caracteres

Una secuencia de caracteres puede lograrse en C++, a través de un apuntador a variables char o un arreglo char, en ambos casos podemos accesar a una serie consecutiva de variables caracter. Como principal característica, las secuencias de caracteres deben de ser terminadas en un caracter nulo, conocido también como NULL, precisamente con el propósito de indicar el final de la secuencia. Dado que sería prácticamente imposible encontrar un teclado con una tecla especial para el caracter nulo, se ha llegado a la conversión de utilizar '\0', para indicar el caracter

NULL. Sin embargo, no siempre es necesario escribir explícitamente el caracter nulo puesto que cuando se denota una cadena de caracteres encerrándose entre comillas dobles "" automáticamente el compilador reserva espacio para los caracteres contenidos entre las comillas, más un caracter NULL para indicar la finalización de la cadena.

La gran diferencia entre usar un apuntador y usar un arreglo de caracteres es que con los arreglos se prevé de antemano el espacio para poder almacenar caracteres, mientras que los apuntadores no tienen asignado una porción de memoria especial para poder ser usados. Un arreglo generalmente se usa cuando se sabe con exactitud la longitud necesaria para el propósito de un programa, de otra forma puede desperdiciarse espacio de memoria que pudiera ser útil en otras actividades. Un apuntador es usado regularmente cuando se desconoce cuál será la longitud requerida para una cadena durante la ejecución de un programa, o cuando dicha longitud es variable. El uso de arreglos regularmente es más sencillo en comparación con el de apuntadores, puesto que el uso indebido del apuntador puede conseguir que se escriba información en partes no deseadas o reservadas de la memoria, con lo que pudiera corromperse el ambiente operativo del computador.

Ejemplos de declaraciones para su uso con secuencias de caracteres están en el Recuadro 6.1, donde las cuatro primeras instrucciones de la función main(), declaran las variables apuntadores a caracteres apCadena1 y apCadena2, así como los arreglos de caracteres aCadena3 y aCadena4. Observe que apCadena1 no apunta a ninguna dirección de memoria hasta que se le asigna la cadena "Ejemplo 3" en la última instrucción de main, mientras que a apCadena2 se le asigna una cadena de caracteres desde el momento mismo en que es creada a través de una inicialización. Los arreglos

aCadena3 y aCadena4 son del mismo tamaño, pero note que a aCadena4 se le ha inicializado caracter por caracter en las primeras nueve localidades con la cadena "Ejemplo 2". Cuando se realiza la inicialización caracter por caracter de una cadena para un arreglo es importante no omitir el caracter nulo. Note que algo muy importante: las cadenas se denotan encerrándose entre comillas dobles "", mientras que un caracter sencillo se encierra entre comillas sencillas ' '.

```
main ()
{
 char *apCadena1,
      *apCadena2     = "Ejemplo 1",
      aCadena3 [30],
      aCadena4 [30] = {'E','j','e','m','p','l', 'o',' ','2', '\0'} ;

 apCadena1 = "Ejemplo 3";

}/*** main ()***/
```

Recuadro 6.1 Ejemplo de cadenas de caracteres

En general puede inicializarse cualquier arreglo durante su declaración a través de la sintaxis:

DeclaraciónDelArreglo = { *Elemento-1*, ...*Elemento-n* };

donde:

DeclaraciónDelArreglo es la definición del arreglo tal y como se ha explicado con anterioridad.

Elemento es un valor asignado a la localidad del arreglo.

por ejemplo, en la inicialización de aCadena4 en el Recuadro 6.1, la primera 'C' se le es asignada a la primera localidad, 'a' se le asigna a la segunda localidad, y así sucesivamente hasta llegar al caracter nulo '\0', al que se le asigna la décima localidad. Por lo tanto para aCadena4 se inicializarán las primeras diez localidades quedando desocupadas las últimas veinte.

6.2. APUNTADORES

Un apuntador en C++, es una variable que contiene la dirección de memoria de otra variable. Por lo tanto, de un apuntador pueden obtenerse dos valores principalmente: el valor del apuntador y el valor al que apunta el apuntador.

El valor del apuntador, es precisamente la dirección que le fue asignada a la variable y basta con el identificador del apuntador para poder obtener su valor. Ejemplos de cómo se maneja el valor de un apuntador es mostrado en la Recuadro 6.2.

Se trata de un programa que intercambia el valor de dos apuntadores. Primero se declaran tres apuntadores, al primero se le asigna el valor nulo en su declaración, mientras que al segundo y al tercero se les asigna una cadena de caracteres. Es muy buena práctica de programación el asignar el valor nulo a todos los apuntadores que no reciben un valor en específico, de esta forma, sabremos que un apuntador no contiene dato alguno si apunta al valor nulo. Dado que un apuntador contiene una dirección de memoria, si se usa con una dirección incorrecta, entonces

```
main ()
{
 char  *apCadena1 = NULL,
       *apCadena2 = "Ejemplo 1 de cadena",
       *apCadena3 = "Ejemplo 2 de cadena";

 apCadena1 = apCadena2;
 apCadena2 = apCadena3;
 apCadena3 = apCadena1;

}/*** main () ***/
```

Recuadro 6.2 Programa ejemplo para el manejo de apuntadores.

seguramente ocurrirá algún error durante la ejecución de un programa, pues se pudiera accesar alguna parte de la memoria y corromper el ambiente operativo del computador. Por ello es que se sugiere la asignación del valor nulo a un apuntador antes de su uso.

El cuerpo programa del Recuadro 6.2, continúa con la asignación del valor de apCadena2 al apuntador apCadena1. En este momento apCadena1, deja de apuntar a NULL y ahora contiene como valor la dirección de memoria en la que se encuentra la cadena "Ejemplo 1 de cadena". Por medio de asignaciones similares termina el programa con el intercambio de los valores del segundo y tercer apuntador a cadenas de caracteres.

En el programa del ejemplo anterior, tan solo trabajamos con el valor del apuntador, es decir, con la dirección de memoria que contenían, y aún cuando intercambiamos las cadenas de caracteres de dos apuntadores, no se hizo ningún acceso al valor al que apuntaban los apuntadores. Para poder hacer referencia al valor al que apunta un apuntador se utiliza el operador asterisco * precediendo al identificador del apuntador.

Un ejemplo de acceso al valor al que apunta un apuntador es mostrado en el Recuadro 6.3. Se trata de una forma bastante rudimentaria y hasta ridícula de cambiar una cadena de caracteres de minúsculas a mayúsculas, sin embargo, para nuestro propósito educativo este ejemplo puede ser bastante provechoso. A la primera letra de la cadena se le asigna 'E' por medio de los paréntesis cuadrados y un índice a manera de arreglo, recuerde que los apuntadores pueden ser tratados como arreglos.

```
main ()
{
 char *apCadena = "ejemplo";

 apCadena [ 0 ] = 'E';
 apCadena++;
 *apCadena      = 'J';
 *++apCadena    = 'E';
 *++apCadena    = 'M';
 *++apCadena    = 'P';
 *++apCadena    = 'L';
 *(apCadena+1)  = '0';
}
```

Recuadro 6.3 Ejemplo de acceso al valor al que apunta un apuntador

Después el apuntador se incrementa en uno. Las dos únicas operaciones aritméticas que se pueden hacer con un apuntador son la suma y la resta, si se incrementa en uno un apuntador, entonces apuntará a la siguiente localidad de memoria. El apuntador en esta instrucción se ha incrementado en uno para acceder al siguiente caracter, puesto que los caracteres de una cadena se encuentran dispuestos de forma consecutiva en la memoria. De esta forma, al terminar el incremento en uno el apuntador apunta a la segunda localidad de la cadena.

En la siguiente instrucción mediante el operador * se asigna el caracter 'J' a la segunda localidad de la cadena. En las siguientes cuatro instrucciones se realiza la misma operación pero simplificada en una misma línea, primero se incrementa el valor del apuntador y después a la localidad a la que apunta el apuntador se le asigna un nuevo caracter.

El ejemplo termina con otra modalidad de acceso al valor al que apunta un apuntador. El apuntador se incrementa en uno con una suma encerrada entre paréntesis y luego, al valor al que apunta la dirección resultante, se le asigna un nuevo caracter. Note que a diferencia de los incrementos anteriores, después de esta operación, el apuntador sigue apuntando a la misma localidad de memoria, en otras palabras su valor no fue alterado puesto que no se le asignó el incremento como ocurrió con las instrucciones anteriores.

6.3. Medir

En diversas ocasiones es requerido el conocer la longitud medida en caracteres de una cadena de caracteres. En el Recuadro 6.4 se puede observar una función que realiza tal tarea llamada `Medir`, esta función recibe un apuntador a una secuencia de caracteres y retorna su longitud en un formato `unsigned long`. La función parece muy sencilla y lo es, dado que asume que la secuencia de caracteres termina en el caracter nulo y aprovecha esto para utilizarlo en la cláusula de comparación `while` en cada ciclo. Esto es porque al llegar a la comparación `while ( *apCadena++ )` se

verifica el contenido de *apCadena y posteriormente se incrementa el apuntador para que apunte a la siguiente localidad de la cadena, de esta forma se tiene listo el apuntador para la comparación del siguiente ciclo. Si *apCadena es igual al caracter nulo entonces deja de ejecutarse el ciclo while de otra forma se incrementa ulLongitud.

```
unsigned long Medir ( char* apCadena )
{
 if ( !apCadena ) return 0;

 unsigned long ulLongitud = 0;

 while ( *apCadena++ ) ulLongitud++;

 return ulLongitud;

} /*** Medir () ***/
```

Recuadro 6.4 La función medir

6.4. EJERCICIOS

6.4.1 *Obligatorio*

Cree un archivo llamado StrUtil.h e inserte en él las funciones descritas en este capítulo.

6.4.2 *Opcional*

Escriba, ejecute y describa el programa llamado medir.cpp que es mostrado en el Recuadro 6.5.

6.4.3 *Obligatorio*

Escriba la función llamada Copiar, cuyo objetivo será copiar un determinado número caracteres de una cadena de caracteres a otra. Recibirá como primer parámetro el apuntador a caracteres destino, como segundo parámetro el apuntador a caracteres fuente y como tercer parámetro el número de caracteres a copiar del apuntador fuente al destino. Retornará un apuntador a la cadena de caracteres destino.

6.4.4 *Opcional*

Escriba el programa copiar.cpp, cuyo objeto sea el comprobar la función Copiar.

```
#include <StrUtil.h>
#include <DepUtil.h>

main ()
{
 int  iContador = 21;
 char *apCadena = "123456789012345678901";

 while ( iContador -- )
 {

  Escribir ( Medir  ( apCadena ) );

  apCadena [iContador-1] = NULL;

 } /*** while ( iContador -- ) ***/

} /*** main () ***/
```

Recuadro 6.5 El programa medir.cpp

6.4.5 *Obligatorio*

Escriba la función `Comparar` que recibirá dos cadenas de caracteres como sus dos primeros parámetros y un caracter como tercer argumento cuyo valor por omisión será el caracter nulo. Esta función retornará un número `int`.

Si la primera cadena de caracteres es menor que la primera, entonces la función retornará un valor negativo. Si la primera cadena es mayor que la primera, entonces la función retornará un valor positivo. Si las dos cadenas son iguales, la función retornará cero.

El caracter del tercer argumento servirá de caracter comodín y podrá ser incluido en cualquiera de las dos cadenas en cualquier número de ocasiones. El comodín funcionará de tal manera que si el caracter comodín se compara contra cualquier otro caracter, entonces se asumirá que se trata de caracteres iguales. Por ejemplo si el caracter comodín es un signo de interrogación, entonces las cadenas "00?110?" y "0001?01" serían tratadas como si fueran la misma.

Inserte la función `Comparar` en el archivo `StrUtil.h`

6.4.6 *Opcional*

Escriba y ejecute un programa que compruebe la función `Comparar`.

6.4.7 *Obligatorio*

Escriba la función `ContarSemejanza` que recibirá dos cadenas de caracteres como parámetros y retornará un número `unsigned long`.

La función retornará el número de caracteres que fueron iguales en posiciones iguales. Por ejemplo, para las cadenas "001101" y "010111", la función retornaría 2 puesto que son iguales los caracteres cuarto y sexto de ambas cadenas.

Introduzca la función ContarSemejanza en el archivo `StrUtil.h`

6.4.8 *Opcional*

Escriba y ejecute un programa que compruebe la función ContarSemejanza.

7. CLREG

Este capítulo tiene como propósito el construir un objeto que se utilice como registro donde guardar datos, tal y como los genes de los cromosomas que usaremos para los algoritmos genéticos. A la definición de tal objeto le llamaremos clReg. Cada objeto clReg estará formado por un arreglo unidimencional de caracteres que será terminado con un caracter nulo. La principal característica de clReg es que todos los objetos de ese tipo serán dinámicos, es decir, el arreglo unidimencional contenido para cada objeto clReg no tendrá una longitud fija o programada, sino determinada a la hora de ejecución del programa.

7.1. DECLARACIÓN DE CLASES

En C++ a la definición de un tipo específico de objeto, se le llama clase. La declaración de una clase sigue la sintaxis general:

```
class NombreDeLaClase
{
  DeclaracionesCaracterísticas
  DeclaracionesAcciones
};
```

donde:

NombreDeLaClase	es un identificador, con el que se podrá hacer posterior referencia a la clase.
DeclaracionesCaracterísticas	son las definiciones de los datos que conforman a una clase.
DeclaracionesAcciones	son los prototipos de las funciones que conforman una clase.

Un ejemplo de la declaración de una clase es mostrada en el Recuadro 7.1. En este ejemplo, es declarada una clase bajo el nombre de clReg, esta clase consta de dos variables, que forman las características, también llamadas *datos miembro* o *variables miembro*. Tales variables son declaradas en sus dos primeras líneas, la primera variable es llamada ulLongReg cuyo tipo es unsigned long, la segunda variable es llamada apReg que es un apuntador a char.

```
class clReg
{
   unsigned long ulLongReg;
   char *apReg;

}; /*** class clReg ***/
```

Recuadro 7.1 Ejemplo de declaración de clase.

7.2. INSTANCIAMIENTO

A final de cuentas, una clase no es más que la definición de un tipo de dato, por lo tanto es posible crear variables a partir de una clase. En C++, a las declaraciones de variables para una clase se les llama objetos y se dice entonces que un objeto es el *instanciamiento* de una clase. La declaración para un objeto sigue exactamente la misma sintaxis de cualquier otra variable:

NombreDeLaClase Objeto(s);

donde:

NombreDeLaClase Es el nombre que identifica a una clase previamente definida.

Objeto(s) Es el identificador del objeto que se creará. En caso de declararse más de un objeto de la misma clase, cada uno de los identificadores se separa por una coma (,).

Un ejemplo de declaración es presentado en el Recuadro 7.2, dentro de la función `main` puede verse en la primera línea la declaración de un objeto, mientras que en la segunda línea aparece la declaración de dos objetos, los tres de tipo `clReg`. Es importante el comprender que cada objeto `clReg` tiene características propias, es decir, al momento de su creación le son asignadas a cada objeto, una variable `unsigned long` y una `char`.

```
class clReg
{
 unsigned long ulLongReg;
 char *apReg;
}; /*** class clReg ***/

main ()
{
 clReg oReg1;
 clReg oReg2, oReg3;

}/*** main ()***/
```

Recuadro 7.2 Ejemplo de instanciamiento de clases

Como se verá en futuras secciones, un objeto puede ser tratado como cualquier otra variable e incluso ser enviado como parámetro a una función.

7.3. EL OPERADOR ::

El operador de cuatro puntos :: del C++, indica pertenencia, del elemento que le sigue a la clase que le antecede. Un elemento es cualquier variable o función. En caso de que ninguna clase preceda al operador ::, entonces se tratará de un elemento global. Un elemento global, es aquel que es declarado fuera de la definición de cualquier clase, no pertenece por lo tanto a ningún objeto y sin embargo puede ser accesado por cualquier objeto.

Ejemplos del uso del operador :: son mostrados en el Recuadro 7.3. Se trata nuevamente de la definición de la clase clReg, solo que ahora a sus dos datos miembro les precede clReg::, lo que significa que pertenecen a la clase clReg. De hecho si se omite clReg:: se asume que los datos pertenecen a la clase por el simple hecho de estar declarados dentro de ella, sin embargo, se ha utilizado este prefijo para ejemplificar su uso.

```
class clReg
{
   unsigned long clReg::ulLongReg;
   char *        clReg::apReg;

}; /*** class clReg ***/
```

Recuadro 7.3 Ejemplos del uso del operador ::.

El operador de pertenencia es usado comúnmente para diferenciar entre acciones y características pertenecientes a las diversas clases que pueden existir, además de las funciones y variables globales.

7.4. ACCESO A LAS CARACTERÍSTICAS DE UN OBJETO

Las características de un objeto pueden ser accesados mediante el operador punto (.). Para accesar una característica se sigue la sintaxis:

Objeto.Característica ;

donde:

Objeto es cualquier instanciamiento de una clase.

Característica es cualquier dato miembro, es decir, cualquier variable declarada para una clase.

7.5. CONSTRUCTORES

Puede definirse una función que se ejecute cada vez que se crea un objeto, a tal función se le llama constructor ó función constructora ó acción constructora. En otras palabras, una función constructora es aquella que es ejecutada al momento de creación de un objeto, esta función se declara llamándose con el mismo nombre de la clase definida. Un ejemplo puede observarse en el Recuadro 7.4, donde se declaran dos acciones para la clase clReg, una de ellas un constructor.

```
class clReg
{
 unsigned long ulLongReg;
 char *apReg;
 inline clReg::ErrorEnConstructor ( void );
 inline clReg::clReg ( const clReg &oIniReg );

}; /*** class clReg ***/
```

Recuadro 7.4 Ejemplos de declaraciones de funciones constructoras y destructoras

Observe que al constructor le precede clReg::, lo que no es necesario, puesto que al igual que pasa con los datos miembro, las funciones miembro pertenecen de forma implícita a la clase en la que están

definidas. Sin embargo en lo que resta, se a preferido hacer explícita la pertenencia de las acciones a sus clases para introducir alguna claridad adicional.

El paso de un objeto como parámetro en una función se hace siempre por valor, lo que implica que sea creado un nuevo objeto para ser utilizado dentro de la función y de esta forma no alterar su valor. Aún cuando esto sea muy cómodo en muchas ocasiones, en otras, puede significar tiempo consumido al momento de creación del objeto. Es por esta razón que generalmente un objeto es pasado por referencia, es decir, como un parámetro variable, añadiéndosele un &, pero para asegurar que el parámetro no será alterado por la función se le antecede con el modificador de tipo const. La cláusula const al momento de declaración de una variable hace que esta se convierta en constante, de tal forma que cualquier intento de modificación a la variable generaría un error al tiempo de compilación. Por ello es que el parámetro de la segunda función constructora llamado oIniReg, es un objeto variable pero inmodificable puesto que le antecede el modificador de tipo const.

El código de la función constructora de clReg es mostrado en el Recuadro 7.5. Se trata del constructor en línea que recibe como parámetro otro objeto tipo clReg llamado oIniReg, el propósito de esta función es el de crear un objeto tipo clReg que tenga los mismos datos que el objeto oIniReg.

Dado que el tamaño de un registro para guardar información puede ser muy variante dependiendo de la aplicación, se ha decidido que cada objeto clReg sea dinámico, es decir, que su tamaño crezca o se reduzca

```
#include <StrUtil.h>

inline clReg::clReg ( const clReg &oIniReg )
{
 if ( (apReg = new char [ 1 + (ulLongReg = oIniReg.ulLongReg) ]) != NULL )
 {
  Copiar ( apReg, oIniReg.apReg, ulLongReg );
  apReg [ulLongReg] = NULL;

 } /*** if ( apReg ) ****/

 else ErrorEnConstructor ();

} /*** clReg::clReg  ( ) ***/
```

Recuadro 7.5 La función constructora clReg a partir de otro objeto clReg

dinámicamente, lo que es muy realizable a través de apuntadores a caracteres y a la instrucción new. Todas las variables que habían sido presentadas hasta el momento son estáticas, esto significa que se reserva un lugar en memoria específico para almacenar su valor, tal lugar en memoria obviamente es fijo y no es alterable su tamaño, en otras palabras no puede ocupar más espacio del que se le había previsto con anterioridad. Sin embargo, puede resultar muy cómodo el crear variables al tiempo de ejecución del programa independientemente de su tipo o longitud, la sentencia new sirve para tal propósito. La orden new crea variables alojadas en localidades consecutivas de memoria y si se tuvo

éxito retorna la dirección de memoria del primer elemento creado ó retorna NULL en caso de fracaso. La sintaxis de new es:

```
VariableTipoApuntador = new Tipo [ NumeroElementos ];
```

donde:

VariableTipoApuntador	Es un apuntador al *Tipo* de variable deseada.
Tipo	Es cualquier tipo de dato.
NumeroElementos	Es el número de variables que se desean crear. Esta parte de la sentencia es opcional y en caso de que se omita se creará una sola variable. Los corchetes [] solo son necesarios si se quiere más de una variable.

por ejemplo, en el Recuadro 7.6 son mostrados varios ejemplos del uso de new. El programa se inicia con la declaración de tres apuntadores uno a entero, uno a caracter y uno a flotante de doble presición respectivamente. Después se utiliza new para crear variables dinámicamente asignándole el resultado a los apuntadores respectivos, así se crea una variable de tipo entero, 30 variables de tipo caracter y 10 variables de tipo flotante de doble precisión. Por último, se asignan valores a los apuntadores que contienen las localidades de las variables creadas para ejemplificar su uso, a la única variable a la que apunta apEntero se le asigna 20. Observe que el uso de *apEntero=20 daría el mismo resultado que apEntero[0]=20. A la cuarta localidad de apCaracter se le asigna '3', finalmente a la octava localidad de apDoble se le asigna 4.2344.

```
main ()
{
 int *apEntero;
 char *apCaracter;
 double *apDoble;

 apEntero = new int;
 apCaracter = new char [30];
 apDoble = new double [10];

 *apEntero = 20;
 apCaracter[5] = '3';
 apDoble [9] = 4.2344;

}/*** main () ***/
```

Recuadro 7.6 Ejemplos de utilización de la instrucción new.

Esta misma técnica con new para crear variables dinámicas es usada en nuestra función constructora del Recuadro 7.4, analicemos la primera instrucción de esta función. Dentro de la cláusula condicional de la sentencia if se usa la variable apReg, definida para uso de la clase clReg. Observe que apReg no está precedido por ningún operador de punto, cuando esto ocurre quiere decir que hay una *pertenencia implícita* lo que significa que la variable pertenece al objeto que invoca a la función, para el caso de la función constructora significaría el objeto que se está creando. A la variable apReg se le asigna en la sentencia if el resultado de una instrucción new que creará una secuencia de caracteres cuya longitud será la misma que oRegIni.ulLongReg más 1 previendo un caracter nulo para terminación de la cadena. Observe que a la vez que se accesa oRegIni.ulLongReg se asigna su valor a la variable ulLongReg del objeto construido, lo que pudiera haberse hecho en otra línea pero prefiere hacerse de esta forma para llegar a una mayor optimización del código pues se requieren menos accesos a las variables. En caso de que la

instrucción new retorne un valor NULL, entonces se ejecutará la función ErrorEnConstructor. Si new no retorna NULL primero se copiará el contenido de oRegIni.apReg a apReg por ulLongReg caracteres, es decir por toda su extensión, por último se le asigna al último caracter de apReg el caracter NULL, con lo que se completa la función constructora.

Es necesario recalcar algo importante, una acción constructora es opcional. De declararse o no un constructor, al crearse un objeto se crearán también las características que lo conforman de forma automática. Un constructor es necesario solo cuando se requiere hacer un proceso inicial con las características del objeto, tal como asignar un valor a sus variables miembro o ejecutar algunas instrucciones. En específico, el uso de un constructor debiera ser obligatorio cuando se usa memoria dinámica para inicializar los apuntadores, tal y como es el caso de la clase clReg.

7.6. ERROR EN CONSTRUCTOR

La función ErrorEnConstructor cuyo prototipo fue declarado en la definición de clReg, es presentada en el Recuadro 7.7. El propósito de esta función es servir de manejador de errores para el caso de que no existiera memoria. Lo que hace es inicializar en valores nulos a las variables del objeto, posteriormente envía un mensaje de error a la salida estándar y termina la ejecución del programa con la función exit().

La función exit() sirve para terminar la ejecución del programa y recibe como parámetro un número que generalmente puede ser reconocido

```
#include <process.h>
#include <iostream.h>

inline void clReg::ErrorEnConstructor ( void )
{
 apReg = NULL;
 ulLongReg = 0;
 cerr << "Memoria insuficiente. Programa abortado";
 exit ( 1 );

} /*** clReg::ErrorEnConstructor ( ) ***/
```

Recuadro 7.7 La función ErrorEnConstructor

en el ambiente operativo desde el cual se ejecuta el programa. La definición de exit() puede ser encontrada dentro de la librería process.h, por ello la instrucción de inclusión de este archivo al inicio de la función.

Observe el uso de cerr que sirve para escribir al dispositivo de error estándar. La forma de utilizar cerr es exactamente igual a la descrita a la de cout.

7.7. DESTRUCTORES

Puede definirse una función que se ejecute cada vez que se elimina un objeto, a tal función se le llama destructor ó función destructora ó acción destructora. En otras palabras, una función destructora es aquella que es ejecutada al momento de eliminación de un objeto, esta función se declara llamándose con el mismo nombre de la clase definida precedida por una tilde (~).

Un ejemplo de declaración de una función destructora para la clase clReg es mostrado en la Recuadro 7.8. Por regla, una función destructora no puede recibir parámetros y no retorna ningún valor.

```
class clReg
{
 unsigned long ulLongReg;
 char *apReg;

 inline void clReg::ErrorEnConstructor ( void );
 inline clReg::clReg ( unsigned long ulLongIniReg = 0 );
 inline clReg::~clReg ( void );

}; /*** class clReg ***/
```

Recuadro 7.8 Ejemplo de la declaración de una función destructora para una clase

La acción destructora definida para clReg puede observarse en el Recuadro 7.9, esta función es verdaderamente pequeña, pero de vital importancia para el correcto funcionamiento de la clase.

```
inline clReg::~clReg ( void )
{
 delete apReg;

} /*** clReg::~clReg ( ) ***/
```

Recuadro 7.9 La función destructora de clReg

El operador delete, realiza exactamente la tarea inversa de new, puesto que libera la memoria que fue previamente obtenida y asignada a un apuntador. La sintaxis de delete es:

delete *Apuntador* ;

donde:

Apuntador es un apuntador a cualquier tipo de dato que contiene la dirección de inicio de una localidad de memoria previamente creada de forma dinámica.

En la función destructora de clReg, se libera la memoria obtenida dinámicamente para apReg, mediante el operador delete. Es importante la liberación de la memoria, puesto que de otra forma pudiese darse el caso de una saturación prematura de la memoria, que de esta manera es evitada.

Es necesario recalcar algo importante, la declaración de la acción destructora para una clase es opcional. De declararse o no un destructor, al eliminarse un objeto se eliminarán también las características que lo conforman de forma automática. Un destructor es necesario solo cuando se requiere hacer un proceso final. En específico, el uso de un destructor debiera ser obligatorio cuando se usa memoria dinámica para liberar la memoria ocupada por los apuntadores, tal y como es el caso de la clase clReg.

7.8. OCULTAMIENTO DE INFORMACIÓN

Dentro de la declaración de una clase, la cláusula private: indica que las declaraciones que siguen son exclusivas del objeto, esto es, utilizables solo por las acciones (funciones miembro) del objeto que las declara, y no son accesibles por otras partes del programa. La cláusula public: indica que las declaraciones que siguen pueden ser accesibles a otras partes del programa. Por omisión todo lo que precede a public: es private:.

En el Recuadro 7.10 se ha incluido un ejemplo de las palabras reservadas public: y private:, de hecho solo se utiliza public:, sin embargo, por omisión todo lo que está declarado antes de public: es private:. Las variables ulLongReg y apReg, así como la función ErrorEnConstructor serán tratados como elementos privados del objeto clReg y por lo tanto, de uso exclusivo para las acciones de clReg e inaccesibles por cualquier otra parte del programa. En este mismo ejemplo, la función constructora y la destructora son las únicas formas de utilizar el objeto, dado que ambas son públicas.

```
class clReg
{
        unsigned long ulLongReg;
        char *apReg;

        inline void clReg::ErrorEnConstructor ( void );
public:
        inline clReg::clReg ( unsigned long ulLongIniReg = 0 );
        inline clReg::~clReg ( void );

}; /*** class clReg ***/
```

Recuadro 7.10 Ejemplo del uso de las cláusulas public: y private:

Como norma programática, es tomado como una buena costumbre de codificación, el mantener dentro de la zona privada a las características junto con acciones para la implantación intrínseca del objeto, mientras que solo son puestas en la zona pública las acciones que pueden tener una utilidad al programador final. De esta forma, tal programador puede hacer funcionar cómodamente un objeto sin distraer su atención para pensar en el cómo es que está formado ó cómo es que funciona un objeto.

Son las cláusulas `private:` y `public:`, formas de ocultar determinados datos y funciones al programador final, de tal suerte que dicho programador no puede accesar más que los elementos del objeto que le son públicos y le quedan ocultos los que son privados. La existencia de elementos privados, permite el que pueda utilizarse un objeto sin uno interesarse en su constitución o funcionamiento, a este paradigma programático se le conoce con el nombre de *abstracción de datos.*

7.9. POLIMORFISMO

En C++, dos o más funciones pueden llamarse con el mismo nombre, siempre y cuando difieran en el número de parámetros, o el tipo de valor retornado, o la clase propietaria. Esta peculiar y útil característica tiene muchas aplicaciones, una de ellas es mostrada en el Recuadro 7.11, donde se tienen dos funciones con el mismo nombre, precisamente las funciones constructoras. La forma de distinguir entre las dos constructoras es que la primera recibe ningún parámetro o un `unsigned long`, mientras tanto, la segunda recibe un objeto tipo `clReg`.

```
class clReg
{
  unsigned long ulLongReg;
  char *apReg;

  inline void clReg::ErrorEnConstructor ( void );
 public:
  inline clReg::clReg ( unsigned long ulLongIniReg = 0 );
  inline clReg::clReg ( const clReg &oIniReg );
  inline clReg::~clReg ( void );

); /*** class clReg ***/
```

Recuadro 7.11 Ejemplo de polimorfismo

La nueva función constructora es una función en línea que recibe como parámetro una variable llamada ulLongIniReg, note que a esta variable le sigue una asignación a cero (=0), esto significa que el valor por omisión de esta variable será cero. En otras palabras, en caso de omitirse un parámetro para esta función, el valor que asumirá la variable será de cero.

El polimorfismo es la cualidad de una función para poder responder a diferentes formas de llamado. El polimorfismo no solo es aplicable a las acciones constructoras de un objeto, sino también a todas sus acciones, puede ser usado también en una función global o hasta en los operadores con significado predefinido en C++, como se verá posteriormente

7.10. ACCESO A LOS CONSTRUCTORES DE UN OBJETO

Para acceder a una función constructora se utiliza el instanciamiento del objeto, siguiendo la siguiente sintaxis si el constructor tiene uno o más parámetros:

```
Clase Objeto ( Parámetro(s) );
```

o también, si el constructor tiene un solo parámetro:

```
Clase Objeto = Parámetro ;
```

o también, si el constructor no tiene parámetros o si todos sus parámetros tienen valores por omisión:

```
Clase Objeto ;
```

donde:

Clase es una clase previamente definida

Objeto es un identificador para el objeto definido.

Parámetro(s) son el(los) argumento(s) requerido(s) para la función constructora

En el Recuadro 7.12, se pueden observar diversos ejemplos de acceso a los constructores de la clase clReg que fue definida en el Recuadro 7.11.

```
main ()
{
 clReg oReg1,
       oReg2 ( 10 ),
       oReg3 = 15,
       oReg4 ( oReg1 ),
       oReg5 = oReg2;

} /*** main () ***/
```

Recuadro 7.12 Ejemplos de acceso a los constructores de un objeto.

Los primeros tres objetos declarados ejecutan al constructor cuyo prototipo tiene como argumento una variable unsigned long. El primer objeto tendrá una longitud de cero pues el constructor asume el parámetro por omisión de cero para su argumento. El segundo objeto ejecuta el constructor con longitud de 10 y el tercero con longitud de 15. Los últimos dos objetos declarados ejecutan al constructor cuyo prototipo tiene como argumento un objeto clReg. Note que en estos ejemplos se puede usar el operador de asignación para ambos constructores porque los dos tienen un solo argumento.

7.11. SOBRECARGA DE OPERADORES

Una extensión del polimorfismo, es lo que se conoce más comúnmente como sobrecarga de operadores, que se refiere a dar un significado diferente a los operadores predefinidos en C++ acorde a las necesidades del programador. Para sobrecargar un operador se crea una función llamada operator seguida por el nombre del operador que se sobrecarga.

Un ejemplo de sobrecarga del operador de asignación es mostrado en el Recuadro 7.13. Puede verse que la función llamada operator= es una función en línea que retorna un objeto clReg, mientras que recibe un parámetro cuyo tipo también es clReg, y que será interpretado como el operando de la derecha del operador de asignación =, esto es, al invocarse un operador sobrecargado, el objeto que invoca la función pasa a ser el operando de la izquierda, mientras que el parámetro pasa a ser el operando de la derecha. Esto se puede observar en el Recuadro 7.14 que al final tiene una función main que declara dos objetos tipo clReg y le asigna a uno el valor del otro en la línea siguiente.

```
class clReg
{
        unsigned long ulLongReg;
        char *apReg;

        inline void  clReg::ErrorEnConstructor ( void );
 public:
        inline clReg::clReg ( unsigned long ulLongIniReg = 0 );
        inline clReg::clReg ( const clReg &oIniReg );
        inline clReg::~clReg ( void );
        inline clReg clReg::operator = ( const clReg &oAsignadoReg );

); /*** class clReg ***/
```

Recuadro 7.13 Ejemplo de sobrecarga del operador de asignación

```
main ()
{
 clReg oReg1, oReg2;

 oReg1 = oReg2;

} /*** main () ***/
```

Recuadro 7.14 Ejemplo de uso del operador = sobrecargado para **clReg**

Uno de los usos más frecuentes de la sobrecarga de operadores es precisamente el operador de asignación, sin embargo, es necesario recordar que puede sobrecargarse cualquier otro operador, ya sea binario como es el caso de la asignación, o unario como sería el caso del incremento.

La función que sobrecarga el operador de asignación esta mostrada en el Recuadro 7.15. Primero por seguridad se le asigna a un apuntador auxiliar llamado apBorradoReg el valor actual de apReg. Después viene una instrucción if en la que ulLongReg toma la longitud del objeto asignado, además de otorgar memoria suficiente a apReg como para almacenar el objeto asignado. Si al otorgarse memoria existe un error se ejecuta la función ErrorEnConstructor, de otra forma se copia el contenido del objeto asignado al actual, poniéndose el caracter nulo al último elemento de apReg. La penúltima instrucción libera la antigua memoria contenida en el apuntador apReg.

Lo más relevante de esta función es el uso de la palabra reservada this dentro del return que es la última sentencia de la acción de asignación. El this es un apuntador al objeto actual, es decir, al objeto que invoca a la

```
#include <StrUtil.h>

inline clReg clReg::operator = ( const clReg &oAsignadoReg )
{
 char *apBorradoReg = apReg;

 if ((apReg = new char [ 1 + (ulLongReg = oAsignadoReg.ulLongReg) ])!=NULL)
 {
  Copiar (apReg, oAsignadoReg.apReg, ulLongReg );
  apReg [ulLongReg] = NULL;

 } /*** if ( apReg ) ****/

 else ErrorEnConstructor ();

 delete apBorradoReg;

 return *this;

} /*** clReg::operator = ( ) ***/
```

Recuadro 7.15 Sobrecaraga del operador de asignación para la clase clReg

función, que para este caso en particular es el objeto al que se le asigna un valor, esto es, el operando de la izquierda en una sentencia de asignación. La palabra this es relativa al objeto que se esté usando y siempre será un apuntador al objeto que invoque la función.

Si vemos nuevamente el prototipo de la función de asignación, notaremos que la función retorna una variable tipo clReg, es decir, un objeto, esto es para dar la apariencia de poder realizar una asignación múltiple, tal y como se puede realizar con el operador de asignación de C++, de tal manera que el objeto retornado por la función pueda ser tomado como parámetro por otra función de asignación como se muestra en el ejemplo del Recuadro 7.16. En este ejemplo se declaran tres objetos, al momento de asignarse el objeto oReg2 manda ejecutar la función de

asignación para asumir las mismas características de oReg3 y al finalizar retorna su mismo valor, que es tomado por oReg1 para realizar el mismo proceso.

```
main ()
{
  clReg oReg1,
        oReg2 (20),
        oReg3 (30);

  oReg1 = oReg2 = oReg3;

} /*** main ()***/
```

Recuadro 7.16 Ejemplo de asignación múltiple

Con ello queda explicado el por qué para finalizar la sentencia de asignación se retorna el valor del objeto con la sentencia return *this.

7.12. EJERCICIOS

7.12.1 *Obligatorio*

Existe una práctica muy común consistente en guardar en un archivo la definición de un objeto, mientras que en otro archivo se guarda la implantación misma del objeto. En adelante para los objetos construidos aquí, almacenaremos la definición del objeto en un archivo con extensión .def, mientras que la implantación será almacenada en un archivo terminado con .imp.

La definición de la clase clReg, es tal y como está en el Recuadro 7.13, introduzca la definición para clReg del Recuadro 7.13 en un archivo llamado clReg.def.

Introduzca el código de las acciones para clReg presentadas en este capítulo, en un archivo llamado clReg.imp.

No olvide hacer la inclusión de los archivos strutil.h, iostream.h, process.h y clReg.imp, en ese orden al final del archivo clReg.def.

7.12.2 *Obligatorio*

Escriba la acción en línea constructora de la clase clReg que recibe como parámetro un número largo sin signo. Esta acción creará una cadena de caracteres cuya longitud está indicada en el parámetro, en caso de no existir error le asignará el caracter de NULL al último elemento de la cadena, de otra forma mandará ejecutar a la función ErrorEnConstructor.

8. clCromosoma

En este capítulo se seguirán introduciendo nociones prácticas importantes del lenguaje C++, mientras que al mismo tiempo se creará la primera clase genética para nuestros algoritmos, la clase clCromosoma.

Tal y como lo propone su nombre, la clase clCromosoma contendrá a los genes de los seres que conforman la población de un algoritmo genético. La clase clCromosoma será muy parecida a la clase clReg, definida con anterioridad, de hecho será su hija. Podemos decir, que la clase clCromosoma se compondrá de un clReg, en cuyas localidades creadas por su apuntador a caracteres se almacenarán los genes, mientras que su variable entera larga servirá para mantener informado del locus máximo del cromosoma. Sin embargo, la clase clCromosoma se compondrá de varias acciones extra que permitirán el acceso con mayor facilidad a cada uno de sus genes, y también proveerá de la simulación de varios operadores que facilitarán la tarea de programación hacia posteriores clases genéticas.

8.1. DEFINICIÓN DE TIPOS

Además de los tipos de datos básicos, dentro del lenguaje C++ pueden crearse nuevos tipos de datos mediante la cláusula typedef cuya sintaxis es:

typedef *TipoPredefinido NuevoTipo* ;

donde:

TipoPredefinido es cualquier tipo de dato ya sea básico, o cualquiera definido con anterioridad mediante otra cláusula typedef.

NuevoTipo es un identificador que sigue las reglas descritas anteriormente para los identificadores y que servirá para hacer referencia al nuevo tipo de dato definido.

Un ejemplo de definiciones de tipos de datos son presentadas en el Recuadro 8.1, donde en la primera línea se declara un nuevo tipo de dato llamado tGen que en realidad es un tipo caracter, posteriormente se define el tipo tCromosoma como un apuntador a genes que en realidad es un apuntador a caracteres. Una vez definidos los nuevos tipos de datos, estos pueden usarse en la definición de variables como si se tratara de cualquier otro tipo de datos, tal y como se muestra dentro de la función main() donde se declaran dos variables tipo tGen y dos tipo tCromosoma. Obviamente, las variables definidas con los nuevos tipos de datos pueden usarse como si se tratase de cualquier otra variable, por ejemplo, las variables tGen pueden ser tratadas como variables caracter y las variables

tCromosoma pueden ser empleadas como cualquier apuntador a cadena de caracteres.

```
typedef char tGen;
typedef tGen *tCromosoma;

main ()
{
 tGen tGen1, tGen2;
 tCromosoma tCrom1, tCrom2;

} /*** main ()***/
```

Recuadro 8.1 Ejemplos de definiciones de nuevos tipos de datos.

Las definiciones de nuevos tipos de datos son muy útiles para aparentar la creación de nuevos tipos de datos, lo que en muchas ocasiones hace más ameno y legible el código de un programa.

8.2. HERENCIA

La herencia, también conocida como derivación de una clase, es la definición de una clase en términos de otras. La herencia consiste en que a través de su declaración, una clase llamada clase hija, obtiene todos los miembros de una o más clases llamadas clases padre. De esta forma, una clase hija constará de los miembros heredados de sus clases padre además de las que la misma clase hija defina. Se dice que hay una herencia simple, o derivación simple de una clase, cuando existe una sola clase padre para una clase hija. Hay una herencia múltiple, o derivación

múltiple de una clase, cuando existen dos o más clases padres para una clase hija. La herencia sigue la sintaxis:

```
class ClaseHija : ClasePadre1, ClasePadre2, ClasePadreN
{
 // definición de la clase hija...
}
```

donde:

ClaseHija Es el identificador con el cual se podrá hacer posterior referencia a la clase.

ClasePadre Son los nombres de las clases de las cuales, la clase definida heredará sus miembros.

Un ejemplo de herencia simple puede observarse en el Recuadro 8.2, donde es declarada la clase clCromosoma que hereda todos los miembros de clReg y además define para sí dos acciones públicas, una de ellas un constructor.

La herencia resulta una práctica de programación bastante rentable, puesto que ahorra código, evita la redefinición de clases, variables y funciones, incrementa las posibilidades de reusabilidad de código, además de permitir conceptualizar fácilmente cualquier desarrollo de software a través de un diseño de *abajo-hacia-arriba*, es decir, partiendo desde pequeños objetos que realicen tareas sencillas y genéricas, hasta objetos complejos que resuelvan problemas complejos y específicos.

```
#include <clReg.def>
typedef char  tGen;
typedef tGen *tCromosoma;

class clCromosoma : clReg
{
public:
 inline clCromosoma::clCromosoma (const clCromosoma &oIniCromosoma );
 inline clCromosoma::clCromosoma (
  unsigned long ulLongCromosoma, tCromosoma tIniCromosoma = NULL
                                 );
 inline clCromosoma clCromosoma::operator = ( const clCromosoma &oAsignadoCromosoma );
 inline clCromosoma clCromosoma::Copiar (
  const clCromosoma &oCromosoma, unsigned long ulLongitud=0,
  unsigned long ulDesplazamiento=0
                                        );
 inline unsigned long clCromosoma::ObtenerLongitud (void);

}; /*** class clCromosoma ***/
```

Recuadro 8.2 Ejemplo de herencia simple

8.3. ACCESO A LOS MIEMBROS HEREDADOS

El heredar una clase de otra hace que la clase hija obtenga todos los miembros de sus padres, es decir todas sus características y acciones. Sin embargo, la clase heredera solo puede tener acceso a los miembros públicos de sus padres. En muchas de las ocasiones, es prudente el hacer que ciertos miembros sean accesibles a las clases herederas pero

inaccesibles a cualquier otra parte del programa, en tal caso se usa la cláusula protected:, cuya sintaxis es:

```
class NombreDeLaClase
{
 public:
 // definición de miembros públicos
 private:
 // definición de miembros privados
 protected:
 // definición de miembros protegidos
 public:
 // definición de miembros públicos
 protected:
 // definición de miembros protegidos
}
```

donde:

NombreDeLaClase Es el identificador con el que se hará posterior referencia a la clase.

Observe de la sintaxis de protected:, que en general las palabras reservadas public:, private: y protected: pueden aparecer en cualquier orden y cualquier número de veces en una clase, particionando estas en múltiples partes privadas, públicas y protegidas. Un ejemplo puede verse en el Recuadro 8.3, que presenta a la antigua clase clReg, con una ligera pero interesante modificación, sus dos características han sido declaradas como privadas de tal manera que serán accesibles a sus clases herederas pero inaccesibles al resto del programa.

El que una clase hija no tenga permiso implícito de acceso a los miembros privados de su padre es debido a que de otra forma todas las

```
class clReg
{
        inline void  clReg::ErrorEnConstructor ( void                          );
 protected:
        unsigned long ulLongReg;
        char          *apReg;
 public:
        inline        clReg::clReg              (unsigned long ulLongIniReg=0);
        inline        clReg::clReg              ( const clReg   &oIniReg      );
        inline        clReg::~clReg             ( void
);
        inline clReg clReg::operator =       ( const clReg   &oAsignadoReg);

}; /*** class clReg ***/
```

Recuadro 8.3 Ejemplo de la cláusula private:

clases quedarían desprotegidas hacia el "mundo" exterior, dado que cualquiera que derivara a una clase podría accesar a sus miembros privados. Las cláusulas protected:, por lo tanto son sumamente útiles en clases genéricas y primitivas cuyo principal objetivo es el ser derivadas, pero esta cláusula es poco recomendable para clases que son hechas con la idea de crear objetos terminales.

8.4. EJECUCIÓN DE CONSTRUCTORES HEREDADOS

Al derivar una clase, si las clases padre tienen funciones constructoras, la clase hija debe tener un constructor en el que se ejecuten los

constructores de sus clases padre. Para ello, al escribir la función constructora de la clase hija se sigue la sintaxis:

```
ClaseHija   ( Parámetros ):
ClasePadre1 ( Parámetros ),
ClasePadre2 ( Parámetros ),
ClasePadreN ( Parámetros )
{
 // Instrucciones para el constructor de la clase hija
}
```

donde:

ClaseHija es el identificador de la clase hija para declarar a su acción constructora

ClasePadre son los identificadores de las clases padre para las que se declara su constructor.

Parámetros son los argumentos respectivos para cada función constructora.

De la anterior sintaxis, es necesario aclarar dos cosas, la primera es que si alguna clase padre no tiene constructores o éstos tienen parámetros implícitos, entonces no es necesario invocarlos desde una función constructora de la clase hija, por lo tanto si ninguna función padre tiene constructores, entonces el constructor de una clase hija sigue la sintaxis de constructores estudiada con anterioridad. La segunda es que los parámetros para las clases padres son tomados de los parámetros de la clase hija, en otras palabras, el constructor de la clase hija debe contener todos los parámetros de los constructores para las clases padres.

Un ejemplo está en el Recuadro 8.4, se trata de uno de los constructores para la clase clCromosoma cuya definición preliminar fue

presentada en el Recuadro 8.2. El constructor recibe dos parámetros, el primero es la longitud del cromosoma y el segundo es un patrón inicial para el cromosoma contenido en una cadena de caracteres. Observe como el primer parámetro de clCromosoma, sirve de parámetro al constructor de clReg. Ya dentro del código de la función, si el parámetro tIniCromosoma es diferente de NULL, entonces se copia en la cadena creada el patrón inicial con la función global Copiar.

```
inline clCromosoma::clCromosoma (
 unsigned long ulLongCromosoma, tCromosoma     tIniCromosoma
                                )
                  :clReg ( ulLongCromosoma )
{
 if (tIniCromosoma) ::Copiar ( apReg, tIniCromosoma, ulLongCromosoma );

} /*** clCromosoma::clCromosoma () ***/
```

Recuadro 8.4 Ejemplo de ejecución de constructores heredados

8.5. CONVERSIÓN A UNA CLASE ANTECESORA

Cualquier objeto de una clase hija puede convertirse a un objeto de cualquiera de sus clases antecesoras mediante el operador de tipificación, tal y como podría tipificarse cualquier otra variable.

Un ejemplo de conversión de una clase hija a una antecesora está en el Recuadro 8.5 que es uno de los constructores para la clase clCromosoma cuya definición ha quedado expuesta en el Recuadro 8.2. Este constructor recibe como parámetro un objeto tipo clCromosoma llamado oIniCromosoma, que ha servido para ejecutar al constructor de clReg a través de una conversión de oIniCromosoma con el tipificador (clReg) con lo que el constructor clReg recibirá como parámetro un objeto clReg. Observe que aún cuando la función no tiene instrucción alguna son necesarios las llaves de inicio y fin de la función.

```
inline clCromosoma::clCromosoma ( const clCromosoma &oIniCromosoma )
                  :clReg ( (clReg) oIniCromosoma )
{
} /*** clCromosoma::clCromosoma () ***/
```

Recuadro 8.5 Ejemplo de conversión de un objeto de clase hija a un objeto de clase antecesora

8.6. EL OPERADOR ->

Como ya se ha descrito anteriormente, el acceso a cualquier miembro de un objeto puede realizarse con el operador punto (.). Sin embargo cuando se tiene un apuntador a una clase, tal y como puede ser el

apuntador implícito this a la clase ejecutora, entonces a los miembros de una clase se les puede accesar la siguiente forma:

Apuntador -> Miembro

donde:

Apuntador Es el identificador de un apuntador a una clase.

Miembro Es el identificador de un miembro de la clase a la que apunta el apuntador

Un ejemplo puede observarse en el Recuadro 8.6, en el que se define la función operator = que fue declarada para la clase clCromosoma en el Recuadro 8.2. Esta función, en su primera instrucción, ejecuta de manera explícita al operador miembro de asignación que fue heredado de la clase clReg. Es necesario destacar varias cosas importantes, la primera es la forma explícita de invocar a la función de asignación mediante el uso de operator= enviando como parámetro un objeto clCromosoma convertido a clReg. La segunda es el uso del operador :: en clReg::, para indicar que se desea llamar al operador de asignación heredado de la clase clReg. La tercera es el uso del operador -> con el apuntador this para accesar al operador de asignación miembro de la clase clReg. La función termina retornando al objeto actual para permitir la apariencia de múltiples asignaciones.

El uso del operador -> es muy usado junto con el apuntador implícito this, sin embargo, su principal uso quizá está durante el acceso a miembros de objetos dinámicos mediante apuntadores a objetos como se verá posteriormente.

```
inline clCromosoma clCromosoma::operator = ( const clCromosoma &oAsignadoCromosoma )
{
 this->clReg::operator=( (clReg) oAsignadoCromosoma );

 return *this;

} /*** clCromosoma::operator = () ***/
```

Recuadro 8.6 Ejemplo de uso del operador ->

8.7. RETORNO DE REFERENCIAS

Una expresión de referencia puede aparecer en ambos lados de una asignación, esto es, asignando su valor o siéndole asignado un valor. Si una función retorna una referencia, entonces puede hacerse una asignación a la llamada de la función, lo que puede ser aprovechado para obtener una valiosa técnica de programación.

Un ejemplo es mostrado en el Recuadro 8.7, en el que se ha hecho una función que sobrecarga al operador [], el parámetro de esta función representa el índice que puede ir dentro de los []. La función retornará una referencia a un caracter del apuntador a la cadena de caracteres heredada, de tal forma que al invocar esta función puede ponérsele delante o detrás de un operador de asignación, simulando de esta forma el uso de arreglos con índices para poder accesar sus elementos.

```
inline char& clCromosoma::operator [] ( unsigned long ulLocalidadCromosoma )
{
 return apReg [ ulLocalidadCromosoma ];
} /*** clCromosoma::operator [] () ***/
```

Recuadro 8.7 Ejemplo del retorno de una referencia

8.8. SOBRECARGA DE OPERADORES PARA FLUJOS

Las formas de escritura y lectura a dispositivos estándar mediante cerr, cout y cin son realizables con el uso de << precisamente porque tanto cerr, cin como cout son objetos mientras que << es un operador sobrecargado para ellos. La clase a la que pertenece cout es ostream, mientras que istream es la clase para cin.

El operador << está sobrecargado de tal forma que retorne una referencia, esto facilita la extensión de la sobrecarga a tipos de datos definidos por el programador final. El operador para la salida se puede sobrecargar con un prototipo de la forma:

ostream& operator << (ostream&, *TipoDefinido*&)

donde:
TipoDefinido es cualquier tipo definido por el programador final

Un ejemplo de este tipo de sobrecarga está mostrado en el Recuadro 8.8, se trata de la sobrecarga para la clase clCromosoma, de tal

forma que pueda escribirse directamente a la salida estándar con el objeto cout, tal y como se haría con cualquier otro tipo de variable.

```
inline ostream& operator << ( ostream &oSalida, const clCromosoma &oCromosoma )
{
 oSalida << oCromosoma.apReg;

 return oSalida;

} /*** clCromosoma::operator << () ***/
```

Recuadro 8.8 Ejemplo de sobrecarga de << para la clase clCromosoma

8.9. FUNCIONES AMIGAS

Una función amiga de una clase, es aquella que no siendo miembro de una clase, tiene acceso a las partes privadas de dicha clase. Para declarar una función amiga a una clase, se incluye el prototipo de la función dentro de la definición de la clase y se le antepone la cláusula friend.

El uso de funciones amigas puede ser provechoso en diversas ocasiones, un ejemplo es mostrado en el Recuadro 8.9, donde el operador de flujos de salida sobrecargado << para la clase clCromosoma es declarado como amigo. El operador tal y como aparece en el Recuadro 8.8 hace acceso a miembros privados de clCromosoma sin ser su miembro, por lo tanto es necesario incluirla dentro de la definición como una función amiga.

```
typedef char  tGen;
typedef tGen *tCromosoma;

class clCromosoma : clReg
{
 public:
 inline clCromosoma::clCromosoma(const clCromosoma &oIniCromosoma);
 inline clCromosoma::clCromosoma(
  unsigned long ulLongCromosoma, tCromosoma tIniCromosoma=NULL
                                );
 inline clCromosoma clCromosoma::operator= ( const clCromosoma &oAsignadoCromosoma );
 inline char& clCromosoma::operator[] ( unsigned long ulLocalidadCromosoma );
 friend ostream& operator<< ( ostream &oSalida, const clCromosoma &oCromosoma );
 inline clCromosoma clCromosoma::Copiar (
  const clCromosoma &oCromosoma, unsigned long ulLongitud  = 0,
  unsigned long ulDesplazamiento = 0
                                        );
 inline unsigned long clCromosoma::ObtenerLongitud (void);

}; /*** class clCromosoma : clReg ***/
```

Recuadro 8.9 Ejemplo de la declaración de una función amiga

Observe que este operador recibe dos parámetros y no uno solo como en las anteriores sobrecargas de operadores. Esto se debe a que esta sobrecarga en particular no pertenece a ninguna clase, sino que es global por lo tanto debe especificarse como primer parámetro el tipo de operando que se espera a la izquierda de operador. Cuando un operador pertenece a una clase solo se utiliza un parámetro que representa al tipo de operando de la derecha, el tipo de operando de la izquierda queda implícito y es tomado el tipo de la clase que sobrecarga el operador.

8.10. EJERCICIOS

8.10.1 *Obligatorio*

Modifique la definición de la clase clReg archivada en clReg.def de tal forma que sea igual a la definición del Recuadro 8.3.

8.10.2 *Obligatorio*

Introduzca el Recuadro 8.9 en un archivo llamado clCromosoma.def, y escriba al final de este archivo la directiva #include <clCromosoma.imp>.

8.10.3 *Obligatorio*

Introduzca del Recuadro 8.4 al Recuadro 8.8 dentro de un archivo llamado clCromosoma.imp.

8.10.4 *Obligatorio*

Programe la acción Copiar para la clase clCromosoma y cuyo prototipo aparece en el Recuadro 8.9. El objetivo de esta función miembro es el copiar el número de genes indicados por ulLongitud del objeto oCromosoma en el objeto actual. La variable ulDesplazamiento indicará la localidad en el objeto actual a partir de la cual se iniciará a copiar el objeto oCromosoma. Inserte esta acción en el archivo clCromosoma.imp.

8.10.5 *Obligatorio*

Programe por usted mismo la acción ObtenerLongitud para la clase clCromosoma cuyo prototipo aparece en el Recuadro 8.9. El objetivo de esta función será retornar la longitud del objeto actual.

8.10.6 *Opcional*

Introduzca el programa de el Recuadro 8.10 en un archivo bajo el nombre de clCromosoma.cpp. Ejecútelo y describa su funcionamiento.

```
#include <clCromosoma.def>
#include <DepUtil.h>
main ()
{
 clCromosoma oCromosoma1 ( 20 ),
             oCromosoma2 = 30,
             oCromosoma3 ( oCromosoma2 ),
             oCromosoma4 = oCromosoma1;
 int iContador = 20;
 while  (iContador -- ) oCromosoma1 [ iContador ] = '0';
 iContador = 30;
 while  (iContador -- ) oCromosoma2 [ iContador ] = '1';
 iContador = 30;
 while  (iContador -- ) oCromosoma3 [ iContador ] = '2';
 iContador = 20;
 while  (iContador -- ) oCromosoma4 [ iContador ] = '3';

 Escribir ( oCromosoma1 );
 Escribir ( oCromosoma2 );
 Escribir ( oCromosoma3 );
 Escribir ( oCromosoma4 );

 oCromosoma3 = oCromosoma1;
 Escribir ( oCromosoma3 );
 oCromosoma3.Copiar ( oCromosoma4, 5, 2 );
 Escribir ( oCromosoma3 );
 oCromosoma3.Copiar ( oCromosoma2, 2 );
 Escribir ( oCromosoma3 );

 oCromosoma3 = oCromosoma1;
 oCromosoma3.Copiar ( oCromosoma2 );
 Escribir ( oCromosoma3 );
} /*** main () ***/
```

Recuadro 8.10 El programa clcromosoma.cpp

9. clGenotipo y clFenotipo

Al momento están en nuestras manos todos los requerimientos para diseñar el fenotipo y genotipo de los seres que formarán la población de un algoritmo genético. El fenotipo estará representado por la clase clFenotipo, mientras que el genotipo es una clase declarada bajo el nombre de clGenotipo.

La clase clGenotipo estará conformada por dos cromosomas uno llamado oCromosomaC, por cromosoma de condición y el otro oCromosomaA, por cromosoma de acción. Se trata de dos tipos de cromosomas para propósito general que están pensados para ser usados de forma indistinta para la optimización de funciones basada en genética, o para el aprendizaje de máquinas basado en genética. Además, clGenotipo estará integrado por constructores que facilitarán su manipulación.

9.1. OBJETOS MIEMBRO

Un objeto miembro es aquel objeto que pertenece a una clase. Un ejemplo de objetos miembro es mostrado en el Recuadro 9.1, donde en la zona protegida de la clase clGenotipo definida, están dos objetos tipo clCromosoma

```
#Include      <clCromosoma.def>

class clGenotipo
{
protected:
clCromosoma oCromosomaC,
            oCromosomaA;

unsigned long ulEspecificidad;

public:
inline clGenotipo::clGenotipo (
 unsigned long ulCromosomaCLong, unsigned long ulCromosomaALong,
 tCromosoma tCromosomaCPatron = NULL, tCromosoma tCromosomaAPatron = NULL
                               );
inline clGenotipo::clGenotipo ( const clGenotipo &oGenotipoIni );

); /*** class clGenotipo ***/

#Include      <clGenotipo.imp>
```

Recuadro 9.1 Ejemplo de declaración de objetos miembro

Tal y como puede imaginarse, un objeto miembro puede utilizarse para simular tanto herencia simple como múltiple. Sin embargo en general se prefiere el uso de la herencia, porque ofrece la posibilidad del uso de la cláusula protected:, además el diseño del código se hace más lógico y entendible. Los objetos miembros se utilizan en situaciones en donde no

se podrían transferir como clases padre, tal y como es el ejemplo anterior, puesto que una clase no se puede heredar dos veces y sin embargo son necesarios dos objetos clCromosoma en la clase clGenotipo.

9.2. EJECUCIÓN DE CONSTRUCTORES MIEMBROS

Cuando se tienen objetos miembros, los constructores de estos se deben ejecutar mediante un constructor de la clase que los contiene. Como en la ejecución de constructores heredados, no es necesario especificar el constructor de un objeto miembro si el constructor no tiene argumentos o si estos tienen valores por omisión. Para ejecutar al constructor de un objeto miembro se sigue la sintaxis:

```
Clase ( Parámetros ):
ObjetoMiembro1 ( Parámetros ),
ObjetoMiembro2 ( Parámetros ),
ObjetoMiembroN ( Parámetros )
{
 // Instrucciones para el constructor de la clase hija
}
```

donde:

Clase	es el identificador de la clase que contiene a los objetos miembro para declarar a su acción constructora
ObjetoMiembro	son los identificadores de las clases padre para las que se declara su constructor.

Parámetros son los argumentos respectivos para cada función constructora.

Tal y como sucede con los constructores heredados, los parámetros de los constructores miembros son tomados de los parámetros del constructor de la clase, de forma que el constructor de la clase contiene todos los parámetros de los constructores para los objetos miembro.

Un ejemplo es mostrado en el Recuadro 9.2, en el que se tiene un constructor a partir de un objeto clGenotipo para la clase del mismo nombre que fue definida en el Recuadro 9.1. Al ser llamado el constructor de la clase se invoca a cada uno de los constructores miembros empezando con oCromosomaC que se construye a partir del objeto oCromosomaC de oGenotipoIni, también oCromosomaA es construido a partir de su homólogo de oGenotipoIni. Finalmente la característica de clGenotipo, ulEspecificidad, es inicializada con su recíproco de oGenotipoIni.

```
inline clGenotipo::clGenotipo  ( const clGenotipo &oGenotipoIni )
                        :oCromosomaC ( oGenotipoIni.oCromosomaC ),
                         oCromosomaA ( oGenotipoIni.oCromosomaA )
{
 ulEspecificidad = oGenotipoIni.ulEspecificidad;

} /*** clGenotipo::clGenotipo () ***/
```

Recuadro 9.2 Ejemplo de ejecución de constructores de objetos miembro

9.3. EJERCICIOS

9.3.1 *Obligatorio*
Introduzca el código del Recuadro 9.1 en un archivo llamado clGenotipo.def.

9.3.2 *Obligatorio*
Introduzca el código del Recuadro 9.2 en un archivo llamado clGenotipo.imp.

9.3.3 *Obligatorio*
Escriba el código para el constructor restante en la definición de clGenotipo del Recuadro 9.1, que recibe dos números enteros largos sin signo y dos variables tCromosoma. El primer número largo sin signo será la longitud del objeto oCromosomaC, el segundo será la longitud de oCromosomaA. La primer variable tCromosoma será el patrón para oCromosomaC y el segundo el de oCromosomaA. Almacene esta acción en clGenotipo.imp.

9.3.4 *Obligatorio*
Escriba la clase clFenotipo cuyo único miembro será una variable public: entera larga sin signo llamada ulFuerza. Guarde esta definición en el archivo clFenotipo.imp.

10. CLSER

Un ser es el elemento básico de un algoritmo genético. Cada ser está integrado por un genotipo y un fenotipo. El genotipo representa información que ha sido generada mediante procesos de simulación genética. El fenotipo representa la adecuación al medio de la información que porta cada ser.

La clase que interpretará a un ser en nuestros algoritmos genéticos será llamada clSer. La clase clSer tiene incorporados diversos mecanismos que pueden ser encontrados en la genética ordinaria, tales como la mutación, la inversión y el entrecruzamiento. Además se encontrarán añadidas varias acciones que permiten el acceso o modificación de las características de cada objeto clSer, así como constructores y operadores que permiten la creación de clones necesarios para facilitar la programación de un algoritmo genético.

La información portada por cada objeto clSer es almacenada en los dos cromosomas de su genotipo llamados cromosoma de acción el uno y cromosoma de condición el otro. Los genes de ambos cromosomas serán binarios, cuyos alelos serán representados por 0 y 1, sin embargo, para el cromosoma de condición se introduce un alelo adicional conocido como

alelo comodín, o llamado también alelo "no importa", que será representado por el -1. El gen comodín, está pensado en facilitar dos finalidades, la de generalización y la de paralelización implícita en el aprendizaje basado en genética.

10.1. DEFINICIÓN

La definición de la clase clSer es expuesta en el Recuadro 10.1, como puede observarse, esta clase es hija de clGenotipo y clFenotipo, por lo tanto heredará todas sus propiedades y tendrá acceso a sus características puesto que fueron declaradas como protegidas y como públicas. La clase clSer, se compone de dos constructores, un operador y diversas acciones, entre las que destacan Entrecruzar, Mutar e Invertir que son los procesos genéticos incorporados en esta clase.

10.2. CONSTRUCTOR DE GENERACIÓN ESPONTÁNEA

La primera generación es aquella creada al iniciar la ejecución de un algoritmo genético y que por lo tanto no tiene antecesores. En otras palabras, dado que al empezar a ejecutar un algoritmo genético no existe población, entonces los seres son creados de forma aleatoria, por ello se dice que no tienen antecesores. Una vez creada la primera generación, los siguientes seres nacerán a partir de la reproducción.

```
#include      <clGenotipo.def>
#include      <clFenotipo.def>

class clSer : clGenotipo , clFenotipo
{
 inline void clSer::Entrecruzar(
  const clSer  &oPadre1, const clSer  &oPadre2,
  float fEntrecruzamientoProbabilidad, unsigned long &ulEntrecruzamientoTotal
                                 );
 void clSer::Mutar ( float fMutacionProbabilidad, unsigned long &ulMutacionTotal );
 void clSer::Invertir (float fInversionProbabilidad,unsigned long &ulInversionTotal);
public:
 clSer::clSer (
  unsigned long ulFuerzaIni, unsigned long ulCromosomaCLong,
  unsigned long ulCromosomaALong
             );
 clSer::clSer (
  clSer &oPadre1, clSer &oPadre2,
  float fEntrecruzamientoProbabilidad, unsigned long &ulEntrecruzamientoTotal,
  float fInversionProbabilidad, unsigned long &ulInversionTotal,
  float fMutacionProbabilidad, unsigned long &ulMutacionTotal,
  tCromosoma tCromosomaCPatron = NULL
             );
 inline clSer::clSer ( const clSer   &oSer  );
 inline clSer clSer::operator=  ( const clSer &oSer );
 inline void clSer::AlterarFuerza ( unsigned long ulAlteracion );
 inline clCromosoma clSer::ObtenerCromosomaA ( void );
 inline clCromosoma clSer::ObtenerCromosomaC ( void );
 inline unsigned long clSer::ObtenerFuerza ( void );

}; /*** class clSer : clGenotipo : clFenotipo ***/

typedef clSer* tApuntadorSer;

#include      <MatUtil.h>
#include      <clSer.imp>
```

Recuadro 10.1 Definición de la clase clSer

Nuestros algoritmos genéticos requieren estrictamente dos seres para que pueda surgir la reproducción, por lo que es importante recalcar que

la cantidad de individuos de la primera generación deberá ser siempre de dos o más seres, para de esta forma, poder crear su prole.

En el Recuadro 10.2 puede observarse un constructor que he denominado *de generación espontanea*, puesto que crea seres sin el requerimiento de otros seres. Por esta peculiaridad este constructor puede utilizarse para obtener una primera generación de la población.

```
clSer::clSer (
 unsigned long ulFuerzaIni, unsigned long ulCromosomaCLong,
 unsigned long ulCromosomaALong
            )
       :clGenotipo  ( ulCromosomaCLong, ulCromosomaALong )
{
 ulFuerza = ulFuerzaIni;
 ulEspecificidad = 0;

 /*** Asignación de genes aleatorios con alelos -1(comodín), 0, 1 ***/
 while ( ulCromosomaCLong -- )
  if ( ( oCromosomaC [ ulCromosomaCLong ] = GenerarAleatorio ( -1, 1 ) ) != -1 )
   ulEspecificidad ++;

 /*** Asignación de genes aleatorios con alelos 0, 1 ***/
 while ( ulCromosomaALong -- ) oCromosomaA [ ulCromosomaALong ]=GenerarAleatorio(0,1);

) /*** clSer::clSer ( ) ***/
```

Recuadro 10.2 El constructor de clSer que simula una generación espontanea

La función recibe como parámetros la fuerza inicial para el clSer creado, y la longitud de los dos cromosomas que lo conforman y que han sido heredados de clGenotipo. Al comenzar, la función inicializa la fuerza y la especificidad. Después hay dos ciclos, en el primero se asignan valores aleatorios entre 1 y -1 al cromosoma de condición, mientras que en el segundo se asignan valores aleatorios entre 1 y 0 al cromosoma de

acción. Note que al momento de inicializar el cromosoma de condición, si el valor es diferente de -1, entonces la especificidad se incrementa, esto quiere decir que la especificidad de un clSer será igual a la longitud del cromosoma de condición menos el número de alelos comodines que contiene.

10.3. ACCESO A LAS ACCIONES DE UN OBJETO

Para accesar a la acción de un objeto, se sigue la sintaxis:

Objeto.Acción (Parámetros)

donde:

Objeto	es cualquier instanciamiento de una clase
Acción	es cualquier función miembro, es decir, cualquier función declarada para una clase.
Parámetros	son los argumentos requeridos para la *Acción*.....

Diversos ejemplos de acceso a las acciones de objetos se podrán observar en las secciones siguientes.

10.4. CONSTRUCTOR DE GENERACIÓN PATERNA

Si se tienen dos o más seres es posible a través de ellos, el crearles descendencia mediante la reproducción. En el Recuadro 10.3 se encuentra un constructor que crea un objeto clSer a partir de otros dos objetos clSer llamados padres, por lo que a este constructor le he denominado *de generación paterna.*

La función constructora recibe como parámetros a los dos padres, la probabilidad de que ocurran entrecruzamiento, inversión y mutación, así como también, en forma de argumento variable, el número de entrecruzamientos, inversiones y mutaciones realizados hasta el momento con el fin de tener a la mano información estadística en caso de requerirse.

El último parámetro del constructor es un patrón de unos y ceros con el cual será inicializado el cromosoma de condición. Este parámetro puede omitirse como puede verse en la definición del Recuadro 10.1, siendo NULL su valor por omisión, en cuyo caso habría cromosoma de acción pero no existiría el cromosoma de condición. El poder omitir al cromosoma de condición permite flexibilidad a nuestros algoritmos para poder acoplarse sin dificultad tanto a problemas de optimización como al aprendizaje de máquinas

En el código de la función, se obtiene la longitud del patrón para el cromosoma de condición y después se comprueba que este valor

```
clSer::clSer (
 clSer &oPadre1, clSer &oPadre2,
 float fEntrecruzamientoProbabilidad, unsigned long &ulEntrecruzamientoTotal,
 float fInversionProbabilidad, unsigned long &ulInversionTotal,
 float fMutacionProbabilidad, unsigned long &ulMutacionTotal,
 tCromosoma  tCromosomaCPatron
           )
    :clGenotipo(
 oPadre1.oCromosomaC.ObtenerLongitud(), oPadre1.oCromosomaA.ObtenerLongitud(),
 tCromosomaCPatron
            )
{
 unsigned long ulCromosomaCLong = oCromosomaC.ObtenerLongitud();
 if (ulCromosomaCLong != Medir ( tCromosomaCPatron ) ) exit ( 2 );

 ulFuerza =(unsigned long)(( oPadre1.ulFuerza + oPadre2.ulFuerza ) / 2);
 ulEspecificidad  = 0;

 /*** Fenómenos cromosómicos ***/
 this->Entrecruzar (
  oPadre1, oPadre2, fEntrecruzamientoProbabilidad, ulEntrecruzamientoTotal
                   );
 this->Invertir ( fInversionProbabilidad, ulInversionTotal );
 this->Mutar ( fMutacionProbabilidad, ulMutacionTotal );

 /*** Conteo de especificidad ***/
 while ( ulCromosomaCLong -- ) if(oCromosomaC[ulCromosomaCLong]!=-1)ulEspecificidad++;

} /*** clSer::clSer ( ) ***/
```

Recuadro 10.3 Constructor de generación paterna para clSer

concuerde con el tamaño de cromosoma del padre, en caso de que sean diferentes, existirá un error y terminará la ejecución del programa. Observe que la fuerza del objeto clSer creado se inicializa con la fuerza promedio de los padres. Posteriormente se invocan los procesos de simulación genética, entrecruzamiento, inversión y mutación, en este orden, para el objeto creado. Por último se calcula la especificidad del

cromosoma de condición mediante un ciclo que incrementa la especificidad por cada alelo diferente de -1.

10.5. ENTRECRUZAMIENTO

Una acción de clSer que realiza el proceso de entrecruzamiento es mostrado en el Recuadro 10.4. Esta función, utiliza los padres enviados como parámetros para copiar de ellos sus cromosomas de acción. También son enviados como parámetros la probabilidad para que se dé un entrecruzamiento, así como un contador que se incrementa en caso de que ocurra el entrecruzamiento.

```
inline void clSer::Entrecruzar (
 const clSer &oPadre1, const clSer &oPadre2,
 float fEntrecruzamientoProbabilidad, unsigned long &ulEntrecruzamientoTotal
                                    )
{
unsigned long ulLocusCruce =
 ((oCromosomaA.ObtenerLongitud()>1) && RealizarEvento(fEntrecruzamientoProbabilidad))
 ? ulEntrecruzamientoTotal++, GenerarAleatorio(1,oCromosomaA.ObtenerLongitud()-1)
 : oCromosomaA.ObtenerLongitud ();

oCromosomaA.Copiar ( oPadre1.oCromosomaA, ulLocusCruce );
oCromosomaA.Copiar (
 oPadre2.oCromosomaA, oCromosomaA.ObtenerLongitud()-ulLocusCruce, ulLocusCruce
                    );

} /*** void clSer::Entrecruzar ( ) ***/
```

Recuadro 10.4 Función de entecruzamiento para clSer

Toda la función se basa en la declaración de la variable entera larga sin signo que contiene el locus de cruce y es inicializada a través de un operador ?: además del operador coma (,). El operador coma evalúa múltiples instrucciones en una sola sentencia de la forma:

Orden1, Orden2, Orden3, OrdenN;

donde:
Orden Es cualquier instrucción válida y ejecutable del lenguaje C++.

Las órdenes del operador coma (,) se evalúan de izquierda a derecha. Por lo tanto, en la instrucción de inicialización del locus de cruce, si se cumple la condición entonces se incrementa el total de entrecruzamiento y se retorna un número aleatorio entre uno y la longitud del cromosoma menos uno, de otra forma, se retorna la longitud del cromosoma de acción. Note como la realización del fenómeno de entrecruzamiento está supeditado, en esta función, por una probabilidad de entrecruzamiento mediante el llamado a la función `RealizarEvento`.

Posteriormente en el método de entrecruzamiento se copian parte de los cromosomas de acción paternos en el hijo, de tal forma que al final de la función es conformado totalmente cromosoma de acción del objeto `clSer` hijo. Note que en caso de que no se realizara el proceso de entrecruzamiento dada su probabilidad, entonces el cromosoma de acción hijo resultante sería igual al cromosoma de acción del primer padre.

10.6. INVERSIÓN

El código para un fenómeno cromosómico de inversión para nuestros algoritmos está en el Recuadro 10.5, como su nombre lo dice la inversión invierte el orden del contenido genético de un cromosoma de tal forma que el gen del locus 1 queda en el último locus, mientras que el último toma el lugar del primero y así sucesivamente para todos los genes del cromosoma.

```
void clSer::Invertir (float fInversionProbabilidad, unsigned long &ulInversionTotal)
{
 if (!(RealizarEvento(fInversionProbabilidad)||(oCromosomaA.ObtenerLongitud()<=1)))
  return;

 /*** Inversión del cromosoma A ***/
 unsigned long ulLocusIni = 0,
               ulCromosomaLong = oCromosomaA.ObtenerLongitud ();
 tGen tGenTemporal;

 while ( ulLocusIni < (--ulCromosomaLong) )
  tGenTemporal = oCromosomaA [ ulLocusIni ],
  oCromosomaA [ ulLocusIni++ ] = oCromosomaA [ ulCromosomaLong ],
  oCromosomaA [ ulCromosomaLong ] = tGenTemporal;

 ulInversionTotal ++;

} /*** void clSer::Invertir ( ) ***/
```

Recuadro 10.5 La acción para el fenómeno de inversión

El cuerpo de la función está basado en el ciclo donde se intercambian los genes hasta que el cromosoma queda totalmente invertido. Note que tal ciclo se realizará bajo la dependencia de la probabilidad de inversión y en caso de que se realice tal proceso entonces se incrementaría el

contador de inversiones. Al igual que el entrecruzamiento, la inversión actúa solamente sobre el cromosoma de acción.

10.7. MUTACIÓN

La mutación de un objeto clSer puede ser posible a través de la acción mostrada en el Recuadro 10.6. A diferencia de los operadores de inversión y entrecruzamiento, el fenómeno de mutación es sometido tanto al cromosoma de condición como al de acción. La mutación para ambos cromosomas se realiza mediante ciclos que recorren locus por locus del cromosoma y de acuerdo a la probabilidad de mutación, la información de un gen puede cambiar a un alelo diferente, en cuyo caso también se incrementa un contador de las mutaciones realizadas.

Para el cromosoma de acción cuyos alelos varían entre 1 y 0, el gen mutado sería igual al valor contrario, es decir, si el alelo actual es 1 entonces el mutado sería 0 y viceversa. La mutación para el cromosoma de condición cuyos alelos varían entre 0, 1 y -1, sería la de cualquiera de los otros alelos restantes, por ejemplo, si el gen a mutar contiene un -1, entonces su mutación posible será 0 o 1.

```
void clSer::Mutar ( float fMutacionProbabilidad, unsigned long &ulMutacionTotal )
{
 unsigned long ulCromosomaALong = oCromosomaA.ObtenerLongitud(),
               ulCromosomaCLong = oCromosomaC.ObtenerLongitud();

 /*** Mutación del cromosoma A ***/
 while ( ulCromosomaALong -- ) if ( RealizarEvento ( fMutacionProbabilidad ) )
  ulMutacionTotal++,
  oCromosomaA [ ulCromosomaALong ] = ( oCromosomaA [ ulCromosomaALong ] + 1 ) % 2;

 /*** Mutación del cromosoma C ***/
 while ( ulCromosomaCLong -- )
  if ( RealizarEvento ( fMutacionProbabilidad ) )
   ulMutacionTotal++,
   oCromosomaC[ulCromosomaCLong]=
    (oCromosomaC[ulCromosomaCLong]+GenerarAleatorio(1,2)+1)%3-1;

} /*** void clSer::Mutar ( ) ***/
```

Recuadro 10.6 La accion de mutación para el objeto clSer

10.8. EJERCICIOS

10.8.1 *Obligatorio*

Introduzca la definición de clSer del Recuadro 10.1 en un archivo llamado clSer.def y al final haga la inclusión a clSer.imp.

10.8.2 *Obligatorio*

Introduzca los constructores de generación espontanea y de generación paterna, así como las acciones de entrecruzamiento, inversión y mutación para el objeto clSer en un archivo llamado clSer.imp.

10.8.3 *Obligatorio*

Escriba un constructor para `clSer` a partir de un objeto `clSer`. El prototipo de este constructor puede encontrarse en el Recuadro 10.1.
Incluya este constructor en el archivo `clSer.imp`.

10.8.4 *Obligatorio*

Escriba un operador de asignación para la clase `clSer`. El prototipo de este operador puede encontrarse en el Recuadro 10.1.
Incluya este operador en el archivo `clSer.imp`.

10.8.5 *Obligatorio*

Escriba una acción llamada `AlterarFuerza` que reciba un parámetro entero largo sin signo y cuyo propósito sea sumar este argumento a la característica `ulFuerza` de la clase `clSer`. El prototipo de esta acción puede encontrarse en el Recuadro 10.1.
Incluya esta acción en el archivo `clSer.imp`.

10.8.6 *Obligatorio*

Escriba las acciones llamadas `ObtenerCromosomaA`, `ObtenerCromosomaC` y `ObtenerLongitud` que retornarán una copia de el cromosoma de acción, una copia del cromosoma de condición y una copia de la longitud de la fuerza de un objeto `clSer`. Los prototipos de estas acciones pueden encontrarse en el Recuadro 10.1.
Incluya esta función en el archivo `clSer.imp`.

11. CLPOBLACION

Un conjunto de seres formarán una población, o dicho en otras palabras, un objeto clPoblacion estará formado por un conjunto de objetos tipo clSer. La clase clPoblacion tiene añadidas acciones básicas que se presentan en una población natural, tales como la reproducción y la muerte, además de la importante función de selección. En este capítulo es explicada con profundidad la estructura de la clase clPoblacion así como su funcionamiento, también se introducen algunos conceptos más acerca de la programación orientada a objetos en el lenguaje C++, que fueron necesarios para el desarrollo de esta clase.

11.1. DEFINICIÓN

La definición de la clase clPoblacion se puede observar en el Recuadro 11.1. Esta definición inicia con la declaración de dFuerzaTotal que contiene la suma del valor absoluto de la fuerza de todos los seres de la población.

```
class clPoblacion
 {
  double dFuerzaTotal;
  float fSemejanzaProbabilidad, fDebilidadProbabilidad;
  unsigned long ulPoblacionTotal, ulPoblacionMax, ulCromosomaCLong, ulCromosomaALong;
  tApuntadorSer *apPoblacion;

  unsigned long clPoblacion::Aglutinar ( tApuntadorSer NuevoSer );
  inline void clPoblacion::Morir ( unsigned long ulSerIndice );
  unsigned long clPoblacion::SeleccionarDebil ( void );

 protected:
  float fMutacionProbabilidad, fEntrecruzamientoProbabilidad, fInversionProbabilidad;
  unsigned long ulEntrecruzamientoTotal, ulInversionTotal, ulMutacionTotal;
  clRegNacimiento oRegNacimiento;

  void clPoblacion::Alojar ( tApuntadorSer apHijo1, tApuntadorSer apHijo2 );
  clSer clPoblacion::SeleccionarFuerte ( unsigned long &ulSerIndice );

 public:
  clPoblacion::clPoblacion (
   unsigned long ulIniCromosomaALong,        unsigned long ulIniCromosomaCLong=0,
   unsigned long ulIniPoblacionMax=0,        unsigned long ulIniPoblacionMin=2,
   float fIniDebilidadProbabilidad=.3,       float fIniSemejanzaProbabilidad=.3,
   float fIniEntrecruzamientoProbabilidad=.8, float fIniInversionProbabilidad=.8,
   float fIniMutacionProbabilidad=.1
                        );
  clPoblacion::~clPoblacion ( void );
  void clPoblacion::Reportar ( void );
  double clPoblacion::ObtenerFuerzaTotal ( void );
  unsigned long clPoblacion::ObtenerPoblacionTotal ( void );
  clSer clPoblacion::ObtenerSer (unsigned long ulSerIndice );
  inline unsigned long clPoblacion::AlterarFuerza (
   unsigned long ulSerIndice, float fAlteracionFuerza
                                                  );
  unsigned long clPoblacion::Reubicar ( unsigned long ulSerIndice );
  virtual void Reproducir ( void ) { };
}; /*** class clPoblacion ***/
```

Recuadro 11.1 Definición para la clase clPoblacion.

Siguen las declaraciones de fSemejanzaProbabilidad y fDebilidadProbabilidad que están estrechamente relacionadas. Al construir

un objeto clPoblacion, es preestablecido un límite máximo de habitantes, si tal límite máximo es rebasado por la realización de una reproducción, entonces es necesaria una rutina de aglutinamiento de la población en la que se hace hueco para un nuevo ser, el hueco, obviamente, se hace a través de la muerte de un ser en la población. Para elegir al ser que morirá es necesario en primera instancia seleccionar a un ser de entre los más débiles de la población, tal y como sucede en los sistemas naturales, pero además, con el fin de mantener la diversidad en la población, se selecciona al ser débil más parecido al nuevo ser que fue producto de la reproducción. El ser débil más parecido al nuevo ser muere entonces y su lugar es ocupado por el nuevo ser. La variable fDebilidadProbabilidad establece la probabilidad de seleccionar al ser más débil en la población, mientras que fSemejanzaProbabilidad marca la probabilidad de seleccionar al ser débil más semejante al nuevo ser.

La variable ulPoblacionTotal contiene el número total de habitantes en la población, mientras que ulPoblacionMax fija el límite máximo de habitantes en la población.

La variables miembro ulCromosomaCLong y ulCromosomaALong contienen respectivamente la longitud del cromosoma C y del cromosoma A, para los objetos clSer que habitan en la población.

La variable apPoblacion, tal vez la más importante de todas, es un apuntador a variables tipo tApuntadorSer, lo que significa que apPoblacion es un apuntador a apuntadores de objetos tipo clSer. Es mediante apPoblacion, que se almacenarán y manipularán todos los habitantes de la población. La técnica usada en el manejo de apPoblación será comprendida

a través de las explicaciones para las acciones que componen a la clase clPoblacion.

Tres de las acciones de clPoblacion son reservadas como privadas, se trata de Aglutinar, Morir y SeleccionarDebil. Para el tratamiento en específico de cada una de ellas es reservada una sección especial en este capítulo.

Continuando con la explicación del Recuadro 11.1, comienza la zona protegida cuyas características y acciones solo serán asequibles por las clases hijas de clPoblacion, tal y como se ha explicado ya con anterioridad. Las variables miembro fInversionProbabilidad, fEntrecruzamientoProbabilidad y fMutacionProbabilidad contienen la probabilidad de ocurrencia de los fenómenos cromosómicos de inversión, entrecruzamiento y mutación respectivamente. Mientras que ulInversionTotal, ulEntrecruzamientoTotal y ulMutacionTotal, contienen el total de fenómenos cromosómicos de inversión, entrecruzamiento y mutación respectivamente, efectuados durante las reproducciones.

Continúa la declaración de un objeto llamado oRegNacimiento que servirá para llevar un registro del natalicio a través de la reproducción. La clase de oRegNacimiento es clRegNacimiento y su definición puede observarse en el Recuadro 11.2 y está formada por siete variables públicas, ulPosPadre1 y ulPosPadre2 son las posiciones que ocupan en la población los dos seres padres; ulPosHijo1 y ulPosHijo2 son las posiciones que ocupan en la población los dos seres hijos resultado de la reproducción; ulPosMuerto1 y ulPosMuerto2 son las posiciones que ocupaban en la población los dos muertos que cedieron su lugar a los seres hijos, pero recuerde que la muerte ocurre solo cuando se llega a la población máxima, por lo que

iSubstitución tendrá un valor diferente de cero cuando se de este caso, e igual a cero cuando no hayan existido muertos al cabo de la reproducción.

```
class clRegNacimiento
{
 public:
        unsigned long ulPosMuerto1,
                      ulPosMuerto2,
                      ulPosPadre1,
                      ulPosPadre2,
                      ulPosHijo1,
                      ulPosHijo2;

        int           iSubstitucion;

}; /*** class sRegNacimiento ***/
```

Recuadro 11.2 Definición de la clase clRegNacimiento.

Siguiendo con la definición de clPoblacion del Recuadro 11.1 podemos ver las acciones Alojar y SeleccionarFuerte con lo que termina la sección privada y comienza la pública, que contiene un constructor y un destructor para la clase, así como a las acciones Reportar, ObtenerFuerzaTotal, ObtenerPoblacionTotal, ObtenerSer, AlterarFuerza y Reubicar. Para las más de estas funciones miembro se ha reservado una sección particular para su estudio, mientras que las restantes serán dejadas en su oportunidad como un ejercicio.

Por último aparece al final de la definición de clPoblacion la acción Reproducir cuyo prototipo es declarado como virtual al inicio, lo que indica que tal acción es virtual, esto significa que la función es miembro de la clase pero su código estará definido por las clases hijas. Por lo

tanto, para escribir el código que define a la acción Reproducir es necesario derivar a la clase clPoblacion y entonces en la clase derivada escribiríamos el código correspondiente a la acción. En general una función se declara como virtual para una clase siguiendo la sintaxis:

virtual *PrototipoDeLaAccion*;

donde:

PrototipoDeLaAccion Es el encabezado de la función miembro definida para una clase.

Una función virtual por sus características es una forma más de simular polimorfismo. Las funciones virtuales resultan de gran utilidad cuando queremos que todas las clases hijas cuenten obligatoriamente con una acción, pero sabemos que la implantación de tal acción puede resultar muy diversa y dependiente para cada clase hija. Con funciones virtuales obligamos a todas las clases hijas a definir una acción con el mismo nombre pero que puede responder diferente (en ocasiones muy diferente) a un mismo mensaje.

Para nuestro caso, en realidad la función de Reproducir puede variar un poco dependiendo de si se trata de la reproducción para un problema de Optimización Basada en Genética o de si la queremos para su uso en Aprendizaje Basado en Genética. Por ello, se ha optado por definirla como virtual y encargar su definición a las clases hijas.

11.2. CONSTRUCTOR

El constructor de clPoblacion es mostrado en el Recuadro 11.3, su objetivo es inicializar todas las variables miembro y crear a una primera generación de seres. Observe como el constructor verifica que las variables fEntrecruzamientoProbabilidad, fInversionProbabilidad, fMutacionProbabilidad, fSemejanzaProbabilidad y fDebilidadProbabilidad estén en rangos válidos de entre 0 y 1, de no ser así se les asigna un valor por omisión.

Los habitantes de una población no pueden ser menos de dos puesto que esto obstruiría el proceso de reproducción. Por otra parte, la población mínima inicial representada por ulIniPoblacionMin no puede ser mayor que la población máxima. Estos aspectos son los que validan las inicializaciones tanto de ulIniPoblacionMin como de ulPoblacionMax. Por otra parte, los contadores que llevan los totales de entrecruzamientos, inversiones, mutaciones y fuerza absoluta son inicializados a ceros.

Observe como se crea suficiente espacio de memoria para albergar a los objetos clSer que conformarán la población a través de la sentencia new aplicada sobre apPoblación. Posteriormente se inicializa el generador de números aleatorios. Por último mediante un ciclo determinado por la población mínima, se inicializa apPoblacion con la creación de objetos clSer en forma dinámica mediante new aplicado sobre la clase clSer invocando al constructor de generación espontánea.

```
clPoblacion::clPoblacion (

  unsigned long ulIniCromosomaALong,                unsigned long ulIniCromosomaCLong,
  unsigned long ulIniPoblacionMax,                  unsigned long ulIniPoblacionMin,
  float         fIniDebilidadProbabilidad,          float    fIniSemejanzaProbabilidad,
  float         fIniEntrecruzamientoProbabilidad,   float    fIniInversionProbabilidad,
  float         fIniMutacionProbabilidad

                                )
{
  ulCromosomaCLong = ulIniCromosomaCLong;
  ulCromosomaALong = ulIniCromosomaALong;

  fEntrecruzamientoProbabilidad = (fIniEntrecruzamientoProbabilidad <= 0)
   || (fIniEntrecruzamientoProbabilidad >= 1) ? .8 : fIniEntrecruzamientoProbabilidad;

  fInversionProbabilidad = (fIniInversionProbabilidad <= 0)
   || (fIniInversionProbabilidad >= 1) ? .8 : fIniInversionProbabilidad;

  fMutacionProbabilidad = (fIniMutacionProbabilidad  <= 0)
   || (fIniMutacionProbabilidad >= 1) ? .9 : fIniMutacionProbabilidad;

  fSemejanzaProbabilidad = (fIniSemejanzaProbabilidad <= 0)
   || (fIniSemejanzaProbabilidad >= 1) ? .3 : fIniSemejanzaProbabilidad;

  fDebilidadProbabilidad = (fIniDebilidadProbabilidad <= 0)
   || (fIniDebilidadProbabilidad >= 1) ? .3 : fIniDebilidadProbabilidad;

  ulPoblacionMax = ulIniPoblacionMax < 2
   ? ulIniCromosomaCLong + ulIniCromosomaALong : ulIniPoblacionMax;

  ulIniPoblacionMin = ulIniPoblacionMin >= 2
   ? ( ulIniPoblacionMin > ulPoblacionMax ? ulPoblacionMax : ulIniPoblacionMin ) : 2;

  apPoblacion             = new tApuntadorSer [ ulPoblacionMax ];

  ulEntrecruzamientoTotal =
  ulInversionTotal        =
  ulMutacionTotal         =
  ulPoblacionTotal        = 0;
  dFuerzaTotal            = 0.0;

  InicializarAleatorios ();

  /*** Nacimiento de la primera generación ***/
  while ( ulIniPoblacionMin-- )
  {
   apPoblacion [ulPoblacionTotal] = new clSer ( ulCromosomaCLong, ulCromosomaALong );
   ulPoblacionTotal++;
  } /*** while () ***/
} /*** clPoblacion::clPoblacion () ***/
```

Recuadro 11.3 Constructor para la clase clPoblacion.

11.3. DESTRUCTOR : *EXTINCIÓN*

En el Recuadro 11.4 está el destructor para la clase clPoblacion, que por sus características lo he llamado *de extinción*. Este destructor extingue la población liberando la memoria ocupada por cada uno de los seres que la habitan y después liberando la memoria misma que le fue asignada dinámicamente a apPoblacion.

```
clPoblacion::~clPoblacion ( )
{
 /*** Extinsión de la población ***/
 while ( ulPoblacionTotal -- ) delete apPoblacion [ ulPoblacionTotal ];

 delete apPoblacion;

} /*** clPoblacion::clPoblacion () ***/
```

Recuadro 11.4 Destructor para la clase clPoblacion

11.4. MUERTE

Como sucede en los sistemas naturales, la muerte puede presentarse para cualquiera de los seres que conforman la población. La función Morir cuyo código es mostrado en el Recuadro 11.5 realiza este proceso.

La función recibe como parámetro ulSerIndice que representa el número del ser dentro de la población que morirá. Antes de morir se resta la fuerza del ser a la fuerza total, de tal forma que esta se mantenga

```
inline void clPoblacion::Morir ( unsigned long ulSerIndice )
{
 /*** Muerte de un ser ***/
 dFuerzaTotal -= fabs ( apPoblacion [ ulSerIndice ]->fFuerza );

 delete apPoblacion [ ulSerIndice ];

} /*** clPoblacion::Morir ( ) ***/
```

Recuadro 11.5 Acción Morir para clPoblacion.

siempre actualizada esta variable. Observe el uso de fabs que retorna el valor absoluto del número de punto flotante enviado como parámetro, fabs es una función estándar definida dentro de la librería math.h. La función finaliza liberando la memoria que ocupaba el objeto clSer mediante la instrucción delete.

11.5. SELECCIÓN : *EL MÁS DÉBIL*

Es cierto que en un sistema natural, la muerte puede presentarse para cualquier ser de la población, sin embargo, es más común o llega más rápido para los seres que debido a su deficiente adecuación al medio han sido poco aptos en su supervivencia. La muerte de los seres más débiles, ha demostrado ser importante para la conservación de las poblaciones naturales, por ejemplo, evita epidemias al eliminar a los enfermos, o preserva alimento y recursos suficientes en la población al suprimir a los más viejos.

Sin embargo, es necesario recalcar que no siempre muere el más débil, si no que existe una escasa, pero latente probabilidad de la muerte de alguno de los fuertes, por lo que la selección natural para la muerte pareciera ser un cruel juego aleatorio, en el que entre más débil es un ser menos probabilidad tiene de subsistir. Pero lejos de ser cruel, la muerte de los débiles es un sacrificio en pro del bien común, del fortalecimiento, optimización y mejor adecuación de la población al medio. Por otra parte, la escasa posibilidad de la muerte de alguno de los fuertes tiene mucha significación, puesto que es un mecanismo para mantener tal diversidad que se pueda responder favorablemente a una diversidad de cambios posibles en el ambiente. Recuerde que la conservación de la especie es dependiente de que tan eficientemente se adecue a un cambiante entorno.

Es la selección natural del más débil, un mecanismo de vital trascendencia que no puede escapar en una simulación que busque aprovechar la robustés de los sistemas naturales. Un proceso que proporciona las bondades de la selección del más débil para la clase clPoblacion se presenta en el Recuadro 11.6.

Primero se selecciona un ser en forma aleatoria de entre la población en la primera declaración de la variable ulSerSeleccionado. Renglones abajo, en la declaración de la variable ulPruebasTotal se calcula el número de pruebas que deberán calcularse sobre la población para buscar al más

```
unsigned long clPoblacion::SeleccionarDebil  ( void )
{

 unsigned long ulSerSeleccionado = GenerarAleatorio ( 0, ulPoblacionTotal-1 ),
  ulPruebasContador = 1,
  ulPruebasTotal = CalcularPruebasRepetidas( ulPoblacionTotal,fDebilidadProbabilidad ),
  ulSerIndice;

 do
  if (
       apPoblacion [ ulSerSeleccionado ]->fFuerza
       > apPoblacion [ulSerIndice=GenerarAleatorio(0,ulPoblacionTotal-1) ]->fFuerza
      ) ulSerSeleccionado = ulSerIndice;
 while ( ++ulPruebasContador < ulPruebasTotal );

 return ulSerSeleccionado;

} /*** clPoblacion::SeleccionarDebil () ***/
```

Recuadro 11.6 La acción SeleccionarDebil para la clase clPoblacion

débil a través de su probabilidad de selección. La selección probabilística se realiza a través de un ciclo do-while, cuya sintaxis es:

```
do
   Instrucciones
while ( Condición );
```

donde:

Instrucciones	Es cualquier instrucción ejecutable en el lenguaje C++.
Condición	Es cualquier expresión aritmética.

Las instrucciones de un ciclo do-while son ejecutadas mientras que la expresión aritmética llamada *Condición* sea diferente de cero. Observe que a diferencia del ciclo while, el ciclo do-while siempre se ejecutará al menos

una vez, esto por que la comparación de la *Condición* es hecha después de que son ejecutadas las *Instrucciones*.

Note en el Recuadro 11.6, que en cada ciclo se selecciona de la población, un ser en forma aleatoria y se compara con el ser seleccionado, en caso de que el aleatorio sea menor que el seleccionado, entonces el seleccionado toma el valor del aleatorio. El ciclo ocurre mientras no se hayan rebasado el número de pruebas repetidas calculadas, y al final del ciclo se retorna la ubicación en la población del ser probabilísticamente más débil.

11.6. SELECCIÓN : *EL MÁS FUERTE*

Es curiosa la existencia de tantos procedimientos previos a la reproducción de las poblaciones en los sistemas naturales, algunos de ellos están llenos de gracia y cortejo, mientras otros no tienen ni lo uno ni lo otro pasando a ser grandes combates en los que se juega la vida. Sin embargo, todos estos procesos y otros más conducen a un solo objetivo: la selección del más fuerte.

La selección del más fuerte es otro de los sustentos genéticos naturales a nivel poblacional que hacen posible la conservación de una especie. Es muy lógico pensar que, el que se reproduzcan los seres más fuertes de una población asegurará que la próxima generación tenga buenas posibilidades de supervivencia dado que heredará las cualidades que

hicieron a los seres paternos los más fuertes por su mejor adecuación al medio.

Al igual que en la selección de los débiles, la naturaleza dicta que la selección de los seres más fuertes para la reproducción debe ser probabilística, de tal forma que los seres débiles tienen una escasa pero existente probabilidad de reproducirse. Esta escasa probabilidad para el más débil es buena, pues es otro factor que permite conservar la diversidad en la población, de tal forma que se pueda responder a la diversidad de cambios que ocurriesen en el ambiente.

Una acción para la clase clPoblacion que simula la selección probabilística del más fuerte está en el Recuadro 11.7. Se trata del algoritmo de bastante uso en las diversas implantaciones de algoritmos genéticos, y es mejor conocido como el método de la ruleta cargada.

```
clSer clPoblacion::SeleccionarFuerte ( unsigned long &ulSerIndice )
{
 float fFuerzaSumatoria = 0,
       fRuletaLocalidad = (float) (GenerarAleatorio ((float)0.0,1.0 )*dFuerzaTotal);

 ulSerIndice = 0;

 do
  fFuerzaSumatoria += fabs ( apPoblacion [ ulSerIndice ++]->fFuerza );
 while ( fFuerzaSumatoria < fRuletaLocalidad );

 return *apPoblacion [ --ulSerIndice ];

} /*** clPoblacion::SeleccionarFuerte () ***/
```

Recuadro 11.7 La acción SeleccionarFuerte para la clase clPoblacion

El algoritmo inicia eligiendo un punto aleatorio en la fuerza total de la población, luego se suman una a una las fuerzas individuales de los seres que conforman la población hasta llegar al punto aleatorio. Resulta lógico pensar que entre mayor sea la fuerza de un ser, más serán sus posibilidades de resultar elegido, en otras palabras la probabilidad de selección de un ser es directamente proporcional a la representación de su fuerza en la fuerza total, por ello el nombre del método de la ruleta cargada.

Observe que al terminar la función SeleccionarFuerte se retorna un objeto clSer mientras que la mismo tiempo, el parámetro variable ulSerIndice contiene la localidad en la población en la que fue localizado probabilísticamente el ser más fuerte.

11.7. REUBICACIÓN

Es una técnica de programación favorable, el mantener ordenada siempre la población a través de la fuerza, de tal forma que el más fuerte se encuentre en la primera posición de la población y el más débil en la última. Esto facilitará en gran medida la obtención posterior de estadísticas, reportes, y por supuesto, la búsqueda de un ser en específico dentro de la población.

Un método de ordenación para los seres de la población es mostrado en el Recuadro 11.8. La función recibe como parámetro un número que representa la localidad en la que se encuentra el ser que se reubicará en

la población. Después la dirección en memoria del ser que será reubicado, se almacena en la variable apSerReubicado.

```
unsigned long clPoblacion::Reubicar ( unsigned long ulSerIndice )
{
 tApuntadorSer apSerReubicado       = apPoblacion[ ulSerIndice ];
 unsigned long ulReubicacionIndice = ulSerIndice,
               ulMaestroIndice      = ulSerIndice,
               ulEsclavoIndice;
 if
  (
   (ulSerIndice > 0
   ?(apPoblacion[ulSerIndice - 1 ]->fFuerza < apPoblacion[ulSerIndice ]->fFuerza):0)
  )
 {
  /*** Busca la localidad del primer clSer mayor hacia arriba ***/
  while
  (
   (ulReubicacionIndice > 0
   ?(apPoblacion[ulReubicacionIndice-1]->fFuerza<apPoblacion[ulSerIndice]->fFuerza):0)
  ) ulReubicacionIndice--;

  ulEsclavoIndice = ulMaestroIndice - 1;
  /*** Recorre a los clSere|s menores hacia abajo ***/
  while(ulReubicacionIndice<ulMaestroIndice)
     apPoblacion [ ulMaestroIndice-- ] = apPoblacion [ ulEsclavoIndice-- ];
 } /*** if ***/
 else
 {
  /*** Busca la localidad del primer clSer menor hacia abajo ***/
  while
  (
   (ulReubicacionIndice < (ulPoblacionTotal-1)
   ?(apPoblacion[ulReubicacionIndice+1]->fFuerza>apPoblacion[ulSerIndice]->fFuerza):0)
  ) ulReubicacionIndice++;

  ulEsclavoIndice = ulMaestroIndice + 1;
  /*** Recorre a los clSere's mayores hacia arriba ***/
  while(ulReubicacionIndice>ulMaestroIndice)
     apPoblacion [ ulMaestroIndice++ ] = apPoblacion [ ulEsclavoIndice++ ];
 } /*** else ***/

 /*** Reubica al ser ***/
 apPoblacion [ ulReubicacionIndice ] = apSerReubicado;
 return ulReubicacionIndice;
} /*** clPoblacion::Reubicar () ***/
```

Recuadro 11.8 La acción Reubicar para la clase clPoblacion

En el `if` que sigue, se pregunta si la localidad actual del objeto `clSer` dentro de la población es mayor a cero y si el ser de la localidad anterior es menor, en caso verdadero, entonces busca hacia arriba de la población un objeto `clSer` mayor o igual y una vez que lo encuentra recorre los seres menores hacia abajo de la población, de tal manera que queda ordenada la población en orden descendente. En caso falso para el `if`, es decir, si la localidad actual del objeto `clSer` es cero o si la fuerza del ser anterior fue mayor, entonces se hace el mismo proceso pero en sentido inverso, es decir, se busca la localidad del primer objeto `clSer` menor hacia abajo de la población y después se recorre a los seres mayores hacia arriba de la población.

La función termina asignando el ser reubicado en la localidad correcta de la población y retorna la nueva posición del ser en la población.

Observe que durante la construcción de la población, no fue necesario hacer reubicación de los seres de la primera generación, puesto que todos ellos surgieron con una misma fuerza inicial, por lo que la primera generación puede darse por ordenada sin necesidad reubicación.

11.8. AGLUTINAMIENTO

En realidad la muerte en la clase `clPoblacion` solo se presentará cuando exista una sobrepoblación, esto es, cuando se sobrepase el límite máximo de habitantes al efectuarse una reproducción que da lugar a un nuevo ser. Cuando esto ocurra es necesario un algoritmo de aglutinamiento para la

población que permita el hacer cupo para el nuevo ser a través de la muerte de un ser de la población. El método más generalmente usado para encontrar al que morirá, es el de elegir de entre los seres más débiles de la población, al más parecido al nuevo ser, de esta forma se asegura no solo que morirá un débil sino que también conservaremos la diversidad en la población.

La técnica de aglutinamiento descrita en el párrafo anterior es presentada en el Recuadro 11.9. La acción recibe como parámetro un apuntador al nuevo ser, es decir, al ser recientemente creado por medio de la reproducción. Después se selecciona probabilísticamente a un ser débil de entre la población y se compara la semejanza de este contra el nuevo ser. En la siguiente línea se calculan el número de pruebas repetidas que se realizarán a fin de encontrar al ser más semejante dentro de los más débiles, dada una probabilidad de semejanza. En un ciclo que se repetirá durante el número de pruebas calculadas, se selecciona otro ser de entre los débiles y si es más semejante al nuevo ser que al seleccionado anterior, entonces se elige como nuevo seleccionado. Al final muere el ser elegido y se retorna la localidad en la que habitaba.

11.9. ALOJAMIENTO

Ante una reciente reproducción es necesario resolver diferentes cuestiones. Primero, si la población máxima es rebasada entonces se debe invocar la rutina de aglutinamiento, depositar al nuevo hijo en la localidad que ocupaba el muerto y posteriormente llamar la función miembro de reubicación para reordenar la población. Pero si la población máxima no

```
unsigned long clPoblacion::Aglutinar ( tApuntadorSer apNuevoSer )
{
 unsigned long ulSerSeleccionado = SeleccionarDebil (),
  ulSemejanzaMax = apNuevoSer->Comparar ( *apPoblacion[ulSerSeleccionado] ),
  ulPruebasTotal=CalcularPruebasRepetidas ( ulPoblacionTotal, fSemejanzaProbabilidad ),
  ulPruebasContador = 1,
  ulSerIndice,
  ulSemejanza;

 do
  if
  (
   ulSemejanzaMax
   <(ulSemejanza=apNuevoSer->Comparar(*apPoblacion[ulSerIndice=SeleccionarDebil()]))
  ) ulSerSeleccionado = ulSerIndice, ulSemejanzaMax = ulSemejanza;
 while ( ++ulPruebasContador < ulPruebasTotal );

 Morir ( ulSerSeleccionado );

 return ulSerSeleccionado;

} /*** clPoblacion::Aglutinar () ***/
```

Recuadro 11.9 La accion Aglutinar para la clase clPoblacion

es rebasada, entonces, se debe depositar al nuevo hijo en una localidad incrementando el número de habitantes, y por su puesto llamando también a la acción de reubicación para mantener ordenada la población.

El Recuadro 11.10 muestra una función que aloja dos nuevos seres enviados como parámetros en la población, note que sigue los pasos planteados en el párrafo anterior. Observe el uso de oRegNacimiento, si existe sobrepoblación oRegNacimiento.iSubstitucion se pone en 1 y se almacenan en ese mismo objeto las posiciones de los dos muertos tras la rutina de aglutinamiento. Si no existe sobrepoblación entonces

oRegNacimiento.iSubstitucion se pone en 0 y el valor de los muertos carece de sentido.

```
void clPoblacion::Alojar ( tApuntadorSer apHijo1, tApuntadorSer apHijo2 )
{
 if ( (ulPoblacionTotal+2) > ulPoblacionMax )
 {
  /*** Aglutina a la poblacisn si hay sobrepoblacisn ***/
  apPoblacion [ ( oRegNacimiento.ulPosMuerto1 = Aglutinar ( apHijo1 ) ) ] = apHijo1;
  oRegNacimiento.ulPosHijo1 = Reubicar ( oRegNacimiento.ulPosMuerto1 );

  apPoblacion [ ( oRegNacimiento.ulPosMuerto2 = Aglutinar ( apHijo2 ) ) ] = apHijo2;
  oRegNacimiento.ulPosHijo2 = Reubicar ( oRegNacimiento.ulPosMuerto2 );

  oRegNacimiento.iSubstitucion = 1;

 } /*** if () ***/

 else
 {
  /*** Incrementa la poblacisn si no hay sobrepoblacisn ***/
  apPoblacion [ ( oRegNacimiento.ulPosHijo1 = ulPoblacionTotal ++ ) ] = apHijo1;
  oRegNacimiento.ulPosHijo1 = Reubicar ( oRegNacimiento.ulPosHijo1 );

  apPoblacion [ ( oRegNacimiento.ulPosHijo2 = ulPoblacionTotal ++ ) ] = apHijo2;
  oRegNacimiento.ulPosHijo2 = Reubicar ( oRegNacimiento.ulPosHijo2 );

  oRegNacimiento.iSubstitucion = 0;

 } /*** else ***/

 dFuerzaTotal += fabs ( apHijo1->fFuerza ) + fabs ( apHijo2->fFuerza );

} /*** clPoblacion::Reproducir ( ) ***/
```

Recuadro 11.10 La acción aglutinar para el objeto clPoblacion

Antes de terminar la fuerza total es incrementada por el valor absoluto de la fuerza de los dos nuevos seres. Note que al final de Alojar, en oRegNacimiento estarán las posiciones actuales de los dos últimos hijos.

Como pudo haberse dado cuenta ya, la posición del primer muerto y del primer hijo pueden variar debido a la acción de reubicación, pero en realidad no es necesario un conocimiento estricto de su localización actual, sino más bien uno aproximado para observar el funcionamiento del algoritmo genético.

11.10. ALTERACIÓN DE LA FUERZA

La fuerza de un ser puede cambiar debido a que tan bien se ha adecuado éste a su medio. Una función miembro para la clase clPoblacion que toma todos los cuidados necesarios en la alteración de la fuerza de uno de los habitantes de la población es mostrado en el Recuadro 11.11.

```
inline unsigned long clPoblacion::AlterarFuerza (
 unsigned long ulSerIndice, float fAlteracionFuerza
                                                  )
{
 dFuerzaTotal -= apPoblacion [ ulSerIndice ]->fFuerza;
 apPoblacion [ ulSerIndice ]->fFuerza = fAlteracionFuerza;
 dFuerzaTotal += fabs (fAlteracionFuerza);
 return Reubicar ( ulSerIndice );

} /*** clPoblacion::AlterarFuerza () ***/
```

Recuadro 11.11 La acción AlterarFuerza para la clase clPoblacion

La acción toma como parámetros, primero el número de ser en la población cuya fuerza será alterada, y en segundo lugar la nueva fuerza para tal ser. El código simplemente decrementa de la antigua fuerza de la fuerza total, le asigna la nueva fuerza al ser deseado e incrementa la

nueva fuerza a la fuerza total de la población. Finalmente reubica al ser al que se le ha cambiado la fuerza para mantener ordenada a la población y retorna el nuevo lugar que ocupa el ser cuya fuerza ha cambiado.

11.11. EJERCICIOS

11.11.1 *Obligatorio*

Introduzca la definición del Recuadro 11.1 para clase clPoblacion en un archivo llamado clPoblacion.def y al final de este incluya el archivo clPoblacion.imp.

Introduzca el código de todas las acciones de la clase clPoblacion en un archivo llamado clPoblacion.imp.

11.11.2 *Obligatorio*

Escriba la función miembro Reportar para la clase clPoblacion, cuyo prototipo aparece en el Recuadro 11.1. Esta acción debe de escribir a la salida estándar un reporte general de la población que contenga los apartados: Parámetros Iniciales, Población, Totales, y Estadísticas.

El apartado Parámetros Iniciales debe contener la longitud del cromosoma de condición, la longitud del cromosoma de acción, la probabilidad de entrecruzamiento, la probabilidad de inversión, la probabilidad de mutación, la probabilidad de semejanza, la probabilidad de debilidad y la población máxima.

El apartado de Población debe de escribir todos los seres que componen a la población.

El apartado de Totales debe contener el total de habitantes en la población, el total de entrecruzamientos realizados, el total de inversiones, y el total de mutaciones.
El apartado de Estadísticas debe contener la fuerza total absoluta, la fuerza máxima, la fuerza mínima y la fuerza promedio absoluta.
Inserte esta función en el archivo clPoblacion.imp.

11.11.3 *Obligatorio*
Escriba la acción ObtenerFuerzaTotal para la clase clPoblacion cuyo prototipo está en el Recuadro 11.1 y que retorna la fuerza total absoluta de la población
Inserte esta función en el archivo clPoblacion.imp.

11.11.4 *Obligatorio*
Escriba la acción ObtenerPoblacionTotal perteneciente a la clase clPoblacion cuyo prototipo está en el Recuadro 11.1 y que retorna el número total de habitantes en la población
Inserte esta función en el archivo clPoblacion.imp.

11.11.5 *Obligatorio*
Escriba la función miembro ObtenerSer cuyo prototipo está en el Recuadro 11.1. Esta acción recibe como parámetro un número que indica la localización de un ser en la población y retorna precisamente a tal ser.
Inserte esta función en el archivo clPoblacion.imp.

12. Optimización Basada en Genética

Este capítulo resulta ser una culminación para el esfuerzo del lector que ha seguido paso a paso lo expuesto con anterioridad, puesto que aquí se desarrolla la primera clase terminal y lista para ser usada. Se trata de la clase clOBG por Optimización Basada en Genética, y como su nombre lo dice, su objetivo es el presentar un esquema con el que se puedan optimizar funciones complejas que se presentan en la vida ordinaria a través de la simulación de procesos genéticos.

12.1. Definición

La definición para la clase clOBG está en el Recuadro 12.1. Se trata de una clase derivada que tiene como padre a la clase clPoblacion, por ende heredará todas sus características y acciones.

Se han definido para clOBG las variables ulContadorGeneracion que contendrá el número de generaciones hasta el momento, ulTotalGeneracion

```
class clOBG : clPoblacion
{
  unsigned long ulContadorGeneracion, ulTotalGeneracion;
  char *apNombreAplicacion;
  float (*ProbarEnElMundo) ( clSer );

  inline void clOBG::Reproducir ( void );
 public:
  inline clOBG::clOBG (
   unsigned long ulLongCromosomaA, unsigned long ulPoblacionMax=0,
   unsigned long ulPoblacionMin=0, float fDebilidadProbabilidad=-1,
   float fSemejanzaProbabilidad=-1, float fEntrecruzamientoProbabilidad=-1,
   float fInversionProbabilidad=-1, float fMutacionProbabilidad=-1
                    );
  clSer clOBG::Optimizar (
   float (*IniProbarEnElMundo) ( clSer ), unsigned long ulIniTotalGeneraciones,
   char* apIniNombreAplicacion="", unsigned uPeriodoReporte=0
                    );

  friend ostream& operator << ( ostream &oSalida, clOBG &oOBG );

}; /*** class clOBG ***/
```

Recuadro 12.1 Definición de la clase clOBG

que indicará el total de generaciones que deben realizarse, apNombreAplicacion que apuntará hacia el nombre de la aplicación que haga uso de la clase, y por ProbarEnElMundo que representa a un nuevo tipo de dato llamado apuntador a una función. Para declarar un apuntador a una función se sigue la sintaxis:

TipoRetornado (* *NombreApuntador*) (*Parámetros*);

donde:

TipoRetornado Es cualquier tipo de dato.
NombreApuntador Es el identificador para el apuntador a una función
Parámetros Es una lista de declaraciones de variables.

El declarar un apuntador a una función, permite que a tal apuntador le pueda ser asignada cualquier función cuyo prototipo concuerde con el que fue declarado el apuntador.

El uso de la función ProbarEnElMundo será uno solo, el de retornar la nueva fuerza del ser que recibe como parámetro, de acuerdo a que tan bien se ha desempeñado en su medio, o que tan buena adecuación al ambiente a mostrado. Su total funcionamiento se entenderá completamente conforme se avance en la lectura en este trabajo.

La variable ProbarElMundo por sus cualidades servirá al objeto clOBG para comunicarse con la aplicación que está haciendo uso de él. Puesto que un apuntador a funciones es capaz de ejecutar cualquier función que siga su prototipo, nos aseguraremos de esta forma que la clase clOBG pueda ser usada para muchos y muy diversos propósitos.

Lo que sigue en el Recuadro 12.1 es la declaración de la acción Reproducir. Como se recordará, este método, fue definido como una función miembro virtual dentro de la clase clPoblacion, lo que obliga a todas sus clases hijas a definir su código en específico para esta acción.

Ya en la parte pública se define un constructor para la clase, también una función llamada Optimizar y se sobrecarga del operador << para la clase ostream de tal forma que se pueda escribir un objeto de esta clase a la salida estándar.

12.2. CONSTRUCTOR

El constructor para la clase clOBG se muestra en el Recuadro 12.2 que como puede notar, es demasiado simple, y su propósito es simplemente invocar al constructor de la clase padre clPoblacion.

```
inline clOBG::clOBG (
 unsigned long ulLongCromosomaA, unsigned long ulPoblacionMax,
 unsigned long ulPoblacionMin, float fDebilidadProbabilidad,
 float fSemejanzaProbabilidad, float fEntrecruzamientoProbabilidad,
 float fInversionProbabilidad, float fMutacionProbabilidad
                    )
            :clPoblacion (
  ulLongCromosomaA, 0, ulPoblacionMax,
  ulPoblacionMin, fDebilidadProbabilidad, fSemejanzaProbabilidad,
  fEntrecruzamientoProbabilidad, fInversionProbabilidad, fMutacionProbabilidad
                          )
{
} /*** clOBG::clOBG () ***/
```

Recuadro 12.2 Constructor para la clase clOBG

12.3. REPRODUCCIÓN

La tan anunciada acción de reproducción puede verse en el Recuadro 12.3. Esta función inicia creando dos objetos tipo clSer que contienen a los dos individuos más fuertes de la población seleccionados probabilísticamente a través de la función SeleccionarFuerte de la clase padre clPoblacion. Note que en las dos llamadas para seleccionar el más fuerte se manda como parámetro oRegNacimiento para que retorne la

ubicación del primer padre y del segundo padre a través de las variables ulPosPadre1 y ulPosPadre2 respectivamente.

```
void clOBG::Reproducir ( void )
{
 /*** Selección probabilística de los dos más fuertes para reproducirse ***/
 clSer oPadre1 = SeleccionarFuerte ( oRegNacimiento.ulPosPadre1 ),
       oPadre2 = SeleccionarFuerte ( oRegNacimiento.ulPosPadre2 );

 /*** Nacimiento de dos nuevos seres ***/
 tApuntadorSer apHijo1 = new clSer (
  oPadre1, oPadre2, fEntrecruzamientoProbabilidad,
  ulEntrecruzamientoTotal, fInversionProbabilidad, ulInversionTotal,
  fMutacionProbabilidad, ulMutacionTotal, NULL
                                    ),

              apHijo2 = new clSer (
  oPadre2, oPadre1, fEntrecruzamientoProbabilidad,
  ulEntrecruzamientoTotal, fInversionProbabilidad, ulInversionTotal,
  fMutacionProbabilidad, ulMutacionTotal, NULL
                                    );

 /*** Alojamiento de los dos nuevos seres ***/
 apHijo1->fFuerza = ProbarEnElMundo( *apHijo1 );
 apHijo2->fFuerza = ProbarEnElMundo( *apHijo2 );

 Alojar ( apHijo1, apHijo2 );

} /*** clPoblacion::Reproducir ( ) ***/
```

Recuadro 12.3 La acción Reproduccir para la clase clOBG

Posteriormente se crean nuevos seres a través del constructor de generación paterna para la clase clSer, observe como para el primer hijo se envía primero al primer padre, mientras que al segundo hijo se le envía primero al segundo padre lo que favorece la diversificación en la

siguiente generación. Los hijos son almacenados en dos apuntadores llamados apHijo1 y apHijo2.

Después de la creación de los dos hijos, se mide su adecuación en el mundo a través del apuntador a funciones ProbarEnElMundo de la clase clOBG. Como puede darse cuenta este apuntador es tratado como si se tratara del identificador de una función cualquiera, enviándosele un objeto tipo clSer y retornando un numero flotante que le es asignado a la fuerza de cada hijo.

La función termina alojando a ambos hijos con lo que también termina el ciclo de reproducción para una generación.

12.4. OPTIMIZACIÓN

La acción Optimización, como su nombre lo dice es el motor para hacer que el algoritmo genético funcione optimizando una función cualquiera. El código de la acción Optimizar de la clase clPoblacion es mostrado en el Recuadro 12.4.

La función recibe como primer parámetro el nombre de una función que retorne un número flotante y que reciba a un objeto clSer, el segundo parámetro indica cuantas generaciones deberán de ejecutarse, el tercero es una cadena de caracteres conteniendo el nombre de la aplicación, y el último es un número que indica cada cuantas generaciones se realizará un reporte a la salida estándar.

```
clSer clOBG::Optimizar (
 float (*IniProbarEnElMundo) ( clSer ), unsigned long ulIniTotalGeneracion,
 char* apIniNombreAplicacion, unsigned uPeriodoReporte
                    )
{
 ProbarEnElMundo = IniProbarEnElMundo;

 /*** Inicializa la primera generación ***/
 unsigned long ulClonPoblacionTotal = ObtenerPoblacionTotal ();
 while  ( ulClonPoblacionTotal -- )
  AlterarFuerza(ulClonPoblacionTotal,ProbarEnElMundo(ObtenerSer(ulClonPoblacionTotal)));

 /*** Ejecución hasta llegar al grado de certeza deseado ***/
 unsigned uContadorReporte = 0;
 ulContadorGeneracion = 0;
 ulTotalGeneracion = ulIniTotalGeneracion;
 apNombreAplicacion = apIniNombreAplicacion;

 while ( ulContadorGeneracion++ < ulTotalGeneracion )
 {
  Reproducir ( );
  if ( uPeriodoReporte && (++uContadorReporte == uPeriodoReporte) )
  {
   cout << *this;
   uContadorReporte = 0;
  } /*** if () ***/
 } /*** while () ***/

 /*** Retorna el clSer más fuerte (óptimo) de la población ***/
 return ObtenerSer ( 0 );

} /*** clRegOBG clOBG::Optimizar () ***/
```

Recuadro 12.4 La acción Optimizar para la clase clOBG

La acción inicia cuando a la variable de la clase clOBG, ProbarEnElMundo, se le asigna el primer parámetro para inicializarla. Después mediante un ciclo también se inicializa la primera generación, recuerde los seres de la primera generación tienen todos una fuerza de cero. Dentro de este ciclo se altera la fuerza de cada uno de los seres creados por generación

espontanea con un llamado a la función AlterarFuerza y ProbarEnElMundo que retornará la fuerza del ser de acuerdo a su adecuación al ambiente.

Un segundo ciclo es ejecutado por el total de generaciones elegido. En él se reproduce la población para crear una nueva generación. Si la variable que contiene el período en que debe realizarse un reporte es diferente de cero, y si un contador indica que se ha llegado tal período, entonces, se ejecuta la escritura del objeto actual a la salida estándar. Como ya se habrá dado cuenta, la escritura del objeto actual a la salida estándar aún no se ha codificado, pues se ha preferido dejar como un ejercicio.

La función termina retornando al ser de la localidad cero en la población, esto es porque la población siempre está ordenada en forma ascendente y en la primera localidad estará el ser más fuerte de la población. De esta forma nos aseguramos de que la optimización terminará retornando al ser más fuerte de la población.

12.5. EJERCICIOS

12.5.1 *Obligatorio*

Escriba una función que sobrecargue el operador << de la clase ostream.h para poder escribir un objeto tipo clOBG a la salida estándar a manera de reporte. El reporte debe contener el nombre de la aplicación, el número de generación, luego debe ejecutar la función Reportar de la clase clPoblacion, y por último debe escribir los resultados de la última reproducción a través de el objeto oRegNacimiento.

12.5.2 *Obligatorio*

Introduzca la definición de la clase clOBG en un archivo llamado clOBG.def.

12.5.3 *Obligatorio*

Introduzca el código de todas las acciones de las clases clOBG, descritas en este capítulo, en un archivo llamado clOBG.imp.

13. EL VIAJERO

Ha llegado el momento de poner a prueba el funcionamiento de nuestra implantación de Optimización Basada en Genética, a través de una aplicación que puede presentarse en la vida real y que se ha convertido en un clásico tema de estudio para diversas disciplinas tales como la Investigación de Operaciones o la Inteligencia Artificial: el viajero.

13.1. PLANTEAMIENTO

Un viajero ha decidido visitar nueve ciudades regresando a la de origen, pero se ha fijado un objetivo, el seguir la ruta que le resulte menos costosa.

Este problema que ha primera vista parece ser de lo más cómodo, resulta ser bastante complejo, puesto que el número de rutas posibles para n ciudades es de $(n-1)!$, lo que significa que para tan solo nueve ciudades podemos elegir entre 40,320 diferentes alternativas.

Intentemos la solución al problema del viajero a través de nuestro algoritmo genético. Primero tenemos que especificar el costo que implicaría viajar de una ciudad a otra. El Recuadro 13.1 muestra una tabla de tipo flotante para nueve ciudades con el costo que implicaría viajar de una ciudad a otra. La tabla se podría leer de la forma, "el costo de viajar de la ciudad 1 a la ciudad 4 es de $43".

```
float fRuta [9] [9] =
{
  0,  1, 23, 43, 12,  4, 23,  4,  9,
  1,  0, 39, 49, 27, 18, 58, 72, 23,
 23, 39,  0, 92, 82, 71, 29, 18, 42,
 43, 49, 92,  0, 47, 68, 10,  2,  4,
 12, 27, 82, 47,  0, 21, 86, 21,  3,
  4, 18, 71, 68, 21,  0,  3,  1,  4,
 23, 58, 29, 10, 86,  3,  0, 12, 23,
  4, 72, 18,  2, 21,  1, 12,  0,  2,
  9, 23, 42,  4,  3,  4, 23,  2,  0
}; /*** fRuta [] ***/
```

Recuadro 13.1 Matriz con los costos de viajes para nueve ciudades

Al escribir esta tabla de costos se ha resuelto casi todo nuestro problema, ahora para ejecutar un objeto de la clase clOBG que optimice los costos de viaje, requerimos de una función que evalúe el rendimiento de los seres creados con el viaje que cada uno de ellos proponen. A esta función de evaluación le llamaremos Viajar.

13.2. VIAJAR

Para idear una técnica de cómo evaluar el rendimiento de un ser, es necesario que pensemos en su carga genética, es decir, en el contenido de sus cromosomas, en específico del cromosoma A, ya que podemos decodificar la información binaria de este cromosoma transformándola en un número decimal. Una vez teniendo el número decimal, podemos pensar en este número obtenido como el número de permutación que se va a evaluar.

Por ejemplo, si la decodificación del cromosoma A de un ser diera 0, entonces estaríamos hablando de la primera permutación, es decir, de la permutación 123456789, lo que significaría que el ser propone como ruta ir a la ciudad 1, luego a la ciudad 2, luego a la ciudad 3, luego a la ciudad 4, luego a la ciudad 5, luego a la ciudad 6, luego a la ciudad 7, luego a la ciudad 8, luego a la ciudad 9 y finalmente de regreso a la ciudad 1. Pero si por ejemplo, la decodificación del cromosoma A de un ser diera 362799, entonces hablaríamos de la última permutación, es decir, de la permutación 987654321. Lo mismo se aplicaría si el ser propusiera cualquiera de las 362800 permutaciones o rutas posibles.

Una vez determinada la ruta propuesta por el ser, el siguiente paso sería acumular el costo total de viajar por esa ruta y retornar el costo. Pero observe que estamos frente a un problema llamado de minimización, puesto que se requieren optimizar los costos, o en otras palabras, minimizar los costos. Por ello, acumularemos el costo del viaje en forma tal que se retorne un número negativo, de manera que la población quede ordenada teniendo siempre en la primera posición al costo más cercano

a cero. Esto es porque, como se recordará, el algoritmo de ordenamiento de la población coloca en la primera posición al más fuerte y en la última al más débil, recuerde también que la función de optimización de la clase clOBG retorna siempre al ser de la primera localidad de la población por considerarlo la mejor opción.

Por ejemplo, si el *ser-1* propone el viaje 237651984 nos daría un costo total de 456, entonces, retornaremos -456. Si el *ser-2* propone el viaje 985674321 nos daría un costo total de 123, entonces retornaríamos -123. De esta forma el algoritmo de ordenamiento de la población pondría al *ser-2* primero que al *ser-1* por tener una fuerza mayor.

El código para la función Viajar, que sigue todas las ideas expuestas en los párrafos anteriores para otorgar la fuerza de un ser de acuerdo con su desempeño o adecuación en el mundo, está en el Recuadro 13.2. La función inicia obteniendo en la variable ulPermutacion, la permutación que propone el ser. La permutación se obtiene con la función Permutar, que fue desarrollada tempranamente en la librería MatUtil.

Hasta este momento en la función se tendría la ruta propuesta del ser, en la variable ulPermutacion. Pero observe que en el ciclo que sigue se va obteniendo número por número el recorrido de la ruta y se va acumulando el costo en la variable uResultado. Al salir del ciclo se calcula el costo del viaje de la última ciudad a la ciudad de partida y por último se retorna el resultado arrojado por la función y que determinará la fuerza de un ser de acuerdo a que tan buena a sido su propuesta de ruta, o en otras palabras, a que tan bien se ha adecuado a los costos impuestos.

```
float Viajar ( clSer oViaje )
{
 float fResultado=0;
 unsigned long ulPermutacion=Permutar(123456789,9,oViaje.ObtenerDecodificacion()+1);
 unsigned uOrigen,
          uDestino,
          uContador=1;

 /*** Calcula el costo de la ruta ***/
 while ( uContador < 9 )
 {
  uOrigen = ObtenerDigito ( ulPermutacion, 9, uContador ) - 1;
  uDestino = ObtenerDigito ( ulPermutacion, 9, ++uContador ) - 1;
  fResultado -= fRuta [ uOrigen] [ uDestino ];

 } /*** while () ***/

 /*** Calcula el costo de regreso al origen ***/
 uOrigen = ObtenerDigito ( ulPermutacion, 9, 9 ) - 1;
 uDestino = ObtenerDigito ( ulPermutacion, 9, 1 ) - 1;
 fResultado -= fRuta [ uOrigen ] [ uDestino ];

 return fResultado;

} /*** float Viajar () ***/
```

Recuadro 13.2 La función Viajar que evalua el rendimiento del viaje propuesto por un ser

13.3. EJECUCIÓN

Un programa que ejecuta y resuelve el problema del viajero puede verse en el Recuadro 13.3. El programa inicia creando un objeto clOBG con una longitud de cromosoma de 9 genes, por las nueve ciudades, para los seres de la población. Después, al mismo tiempo que se declara un objeto clSer, se le inicializa con el resultado de la acción de optimización del objeto oViajes. Observe que el primer parámetro de Optimizar es un

identificador de función, precisamente el nombre de la función Viajar. Los siguientes parámetros son 50 para el número de generaciones, el nombre de la aplicación y por último un reporte cada 10 generaciones.

```
main ()
{
  clOBG oViajes ( 9 );
  clSer oViajeOptimo = oViajes.Optimizar (
                          Viajar, 50, ": Problema del viajero para 9 ciudades", 10
                                );

  Detener (" El ser de la población que propone el viaje menos costoso es..." );
  Escribir ( oViajeOptimo );
  Detener (" La ruta menos costosa es ..." );
  Escribir ( Permutar ( 123456789, 9, oViajeOptimo.ObtenerDecodificacion () ) );
  Detener (" El costo de la ruta prouesta es ... " )
  Escribir ( Viajar (oViajeOptimo ) );

} /*** main () ***/
```

Recuadro 13.3 La función principal que ejecuta y resuelve el problema del viajero

Al terminar la optimización se escribirá el ser que propone el viaje menos costoso, después se escribirá la ruta menos costosa y finalmente se escribirá el costo del viaje óptimo.

Los cinco reportes para el programa anterior son listados del Recuadro 13.4 al Recuadro 13.8. Observe como de generación en generación va disminuyendo la fuerza promedio, lo que significa que se están logrando costos más bajos para el problema del viajero. Finalmente se presenta la solución retornada por el programa en el Recuadro 13.9. En estos listados y con los ejercicios propuestos al final del capítulo podrá darse cuenta de la verdadera potencia de un algoritmo genético. Juzgue Usted mismo...

```
Optimización Basada en Genética
Reporte : Problema del viajero para 9 ciudades
Generación 10 de 50

Parámetros Iniciales:
              Longitud de CromosomaA = 9
   Probabilidad de Entrecruzamiento = 0.9
          Probabilidad de Inversión = 0.8
           Probabilidad de Mutación = 0.400516
          Probabilidad de Semejanza = 0.3
          Probabilidad de Debilidad = 0.75
                    Población Máxima = 9

Totales:
                                Habitantes = 9
                        Entrecruzamientos = 17
                                Inversiones = 12
                                 Mutaciones = 76

Estadísticas:
              Fuerza Total (absoluta) = 1477
                           Fuerza Máxima = -115
                           Fuerza Mínima = -336
          Fuerza Promedio (absoluta) = 164.111111

Población:
   1)A=101110111 D=375 F=-115
   2)A=101110111 D=375 F=-115
   3)A=110101100 D=428 F=-115
   4)A=011100001 D=225 F=-148
   5)A=101101010 D=362 F=-157
   6)A=111000011 D=451 F=-159
   7)A=010011000 D=152 F=-166
   8)A=011010100 D=212 F=-166
   9)A=000101111 D=47 F=-336

Resultados de la última reproducción:
          Posición actual del Hijo 1 = 8
          Posición actual del Hijo 2 = 9
        Posición previa del Padre 1 = 3
        Posición previa del Padre 2 = 6
       Posición previa del Muerto 1 = 9
       Posición previa del Muerto 2 = 9
```

Recuadro 13.4 Reporte de clOBG para el problema el programa VIAJERO después de 10 generaciones.

```
Optimización Basada en Genética
Reporte : Problema del viajero para 9 ciudades
Generacion 20 de 50

Parámetros Iniciales:
            Longitud de CromosomaA = 9
  Probabilidad de Entrecruzamiento = 0.9
          Probabilidad de Inversión = 0.8
          Probabilidad de Mutación = 0.400516
          Probabilidad de Semejanza = 0.3
          Probabilidad de Debilidad = 0.75
                    Población Máxima = 9

Totales:
                                Habitantes = 9
                    Entrecruzamientos = 34
                            Inversiones = 29
                            Mutaciones = 149

Estadísticas:
              Fuerza Total (absoluta) = 1400
                          Fuerza Máxima = -111
                          Fuerza Mínima = -320
        Fuerza Promedio (absoluta) = 155.555556

Población:
  1)A=101111001 D=377 F=-111
  2)A=101110111 D=375 F=-115
  3)A=101110111 D=375 F=-115
  4)A=110101100 D=428 F=-115
  5)A=011100001 D=225 F=-148
  6)A=011011101 D=221 F=-158
  7)A=111000011 D=451 F=-159
  8)A=110011100 D=412 F=-159
  9)A=011111101 D=253 F=-320

Resultados de la última reproducción:
        Posición actual del Hijo 1 = 8
        Posición actual del Hijo 2 = 9
      Posición previa del Padre 1 = 2
      Posición previa del Padre 2 = 1
     Posición previa del Muerto 1 = 9
     Posición previa del Muerto 2 = 9
```

Recuadro 13.5 Reporte de clOBG para el problema el programa VIAJERO después de 20 generaciones

```
Optimización Basada en Genética
Reporte : Problema del viajero para 9 ciudades
Generacion 30 de 50

Parámetros Iniciales:
              Longitud de CromosomaA = 9
  Probabilidad de Entrecruzamiento = 0.9
          Probabilidad de Inversión = 0.8
            Probabilidad de Mutación = 0.400516
          Probabilidad de Semejanza = 0.3
          Probabilidad de Debilidad = 0.75
                    Población Máxima = 9

Totales:
                                Habitantes = 9
                      Entrecruzamientos = 52
                              Inversiones = 43
                               Mutaciones = 229

Estadísticas:
             Fuerza Total (absoluta) = 1245
                          Fuerza Máxima = -111
                          Fuerza Mínima = -209
        Fuerza Promedio (absoluta) = 138.333333

Población:
   1)A=101111001 D=377 F=-111
   2)A=101110111 D=375 F=-115
   3)A=101110111 D=375 F=-115
   4)A=110101100 D=428 F=-115
   5)A=101111110 D=382 F=-121
   6)A=011100001 D=225 F=-148
   7)A=110011111 D=415 F=-153
   8)A=011011101 D=221 F=-158
   9)A=010000100 D=132 F=-209

Resultados de la última reproducción:
         Posición actual del Hijo 1 = 9
         Posición actual del Hijo 2 = 9
       Posición previa del Padre 1 = 1
       Posición previa del Padre 2 = 2
      Posición previa del Muerto 1 = 9
      Posición previa del Muerto 2 = 9
```

Recuadro 13.6 Reporte de clOBG para el problema el programa VIAJERO después de 30 generaciones

```
Optimización Basada en Genética
Reporte : Problema del viajero para 9 ciudades
Generacion 40 de 50

Parámetros Iniciales:
              Longitud de CromosomaA = 9
  Probabilidad de Entrecruzamiento = 0.9
          Probabilidad de Inversión = 0.8
           Probabilidad de Mutación = 0.400516
          Probabilidad de Semejanza = 0.3
          Probabilidad de Debilidad = 0.75
                    Población Máxima = 9

Totales:
                                Habitantes = 9
                    Entrecruzamientos = 68
                            Inversiones = 56
                             Mutaciones = 318

Estadísticas:
            Fuerza Total (absoluta) = 1207
                        Fuerza Máxima = -94
                        Fuerza Mínima = -222
        Fuerza Promedio (absoluta) = 134.111111

Población:
  1)A=110100110 D=422 F=-94
  2)A=101111001 D=377 F=-111
  3)A=101110111 D=375 F=-115
  4)A=101110111 D=375 F=-115
  5)A=110101100 D=428 F=-115
  6)A=101111110 D=382 F=-121
  7)A=110101110 D=430 F=-145
  8)A=111001010 D=458 F=-169
  9)A=010000110 D=134 F=-222

Resultados de la última reproducción:
         Posición actual del Hijo 1 = 9
         Posición actual del Hijo 2 = 9
       Posición previa del Padre 1 = 2
       Posición previa del Padre 2 = 6
      Posición previa del Muerto 1 = 9
      Posición previa del Muerto 2 = 9
```

Recuadro 13.7 Reporte de clOBG para el problema el programa VIAJERO después de 40 generaciones

```
Optimización Basada en Genética
Reporte : Problema del viajero para 9 ciudades
Generacion 50 de 50

Parámetros Iniciales:
              Longitud de CromosomaA = 9
  Probabilidad de Entrecruzamiento = 0.9
          Probabilidad de Inversión = 0.8
            Probabilidad de Mutación = 0.400516
          Probabilidad de Semejanza = 0.3
          Probabilidad de Debilidad = 0.75
                    Población Máxima = 9

Totales:
                              Habitantes = 9
                    Entrecruzamientos = 85
                          Inversiones = 73
                           Mutaciones = 379

Estadísticas:
             Fuerza Total (absoluta) = 1212
                        Fuerza Máxima = -94
                        Fuerza Mínima = -246
        Fuerza Promedio (absoluta) = 134.666667

Población:
  1)A=110100110 D=422 F=-94
  2)A=110100011 D=419 F=-106
  3)A=101111001 D=377 F=-111
  4)A=101110111 D=375 F=-115
  5)A=101110111 D=375 F=-115
  6)A=101101100 D=428 F=-115
  7)A=110101110 D=430 F=-145
  8)A=011100011 D=227 F=-165
  9)A=011001101 D=205 F=-246

Resultados de la última reproducción:
         Posición actual del Hijo 1 = 9
         Posición actual del Hijo 2 = 9
       Posición previa del Padre 1 = 1
       Posición previa del Padre 2 = 1
      Posición previa del Muerto 1 = 9
      Posición previa del Muerto 2 = 9
```

Recuadro 13.8 Reporte de clOBG para el problema el programa VIAJERO después de 50 generaciones

```
El ser de la población que propone el viaje menos costoso es ...
oViajeOptimo = A=110100110 D=422 F=-94

La ruta menos costosa es ...
Permutar ( 123456789, 9, oViajeOptimo.ObtenerDecodificacion () ) = 123768495

El costo de la ruta propuesta es ...
Viajar (oViajeOptimo ) = -94
```

Recuadro 13.9 Solución final del programa VIAJERO al problema del viajero para nueve ciudades

13.4. EJERCICIOS

13.4.1 *Obligatorio*

Introduzca todo el código expuesto en el capítulo en un archivo bajo el nombre de viajero.cpp. Ejecute el programa tal y como se ha propuesto. ¿Han sido iguales los resultados a los mostrados del Recuadro 13.4 al Recuadro 13.8 ?, ¿Porqué?

13.4.2 *Sugerencia*

Para los siguientes ejercicios haga más corto el período del reporte para un mejor apreciamiento de su funcionamiento y cambios.

13.4.3 *Opcional*

Ejecute varias veces el programa iniciando con una población máxima de 10 e incrementándola de 10 en 10 hasta 50. Luego empiece con una

población máxima de 8 y disminúyala de uno en uno hasta 4. ¿Cómo afecta el tamaño de la población en el resultado?, ¿Porqué?

13.4.4 *Opcional*

Ejecute varias veces el programa manteniendo fijas las probabilidades de inversión y mutación en .5, e incremente en una décima por ejecución a la probabilidad de entrecruzamiento iniciando con .5 y terminando en .99 ¿Cómo afectan estos cambios al resultado?, ¿Porqué?

13.4.5 *Opcional*

Ejecute varias veces el programa manteniendo fijas las probabilidades de entrecruzamiento y mutación en .5, e incremente en una décima por ejecución a la probabilidad de inversión iniciando con .5 y terminando en .99 ¿Cómo afectan estos cambios al resultado?, ¿Porqué?

13.4.6 *Opcional*

Ejecute varias veces el programa manteniendo fijas las probabilidades de inversión y entrecruzamiento en .5, e incremente en una décima por ejecución a la probabilidad de mutación iniciando con .5 y terminando en .99 ¿Cómo afectan estos cambios al resultado?, ¿Porqué?

13.4.7 *Opcional*

Ejecute varias veces el programa manteniendo fija la probabilidad de semejanza en .3 e incrementando en una décima por ejecución a la probabilidad de debilidad iniciando con .01 y terminando en .99 ¿Cómo afectan estos cambios al resultado?, ¿Porqué?

13.4.8 *Opcional*

Ejecute varias veces el programa manteniendo fija la probabilidad de debilidad en .3 e incrementando en una décima por ejecución a la probabilidad de semejanza iniciando con .01 y terminando en .99 ¿Cómo afectan estos cambios al resultado?, ¿Porqué?

13.4.9 *Opcional*

Ejecute el programa con una población mínima inicial de 2 habitantes. ¿Cómo afecta el tener una población inicial pequeña en el resultado?, ¿Porqué?

14. CONCLUSIONES

Cuando comentaba con personas cercanas a mí, que me encontraba trabajando con algoritmos genéticos las reacciones eran muchas y muy diferentes. Hubo quien no dudo en que resultaría y se mostraba maravillado con la gran cantidad de aplicaciones posibles. Hubo quien escuchaba con atención y perplejidad otorgándome cierto grado de credibilidad. Hubo quien me miraba con extrañeza y delineaba una cierta sonrisa. Pero hubo un amigo, al que tengo en gran estima, quien me confundió con un biólogo genetista (al menos eso quiero creer), y me dijo "... pero como que reproducción, muerte, selección, mutaciones, cruces y todas esas cosas, bueno pero qué pasa, te estas volviendo loco o qué ...".

No me extrañaría que al iniciar la lectura de este trabajo, nuestro estimado lector estuviera de acuerdo con una o muchas de las opiniones del párrafo anterior. Pero ahora que resta tan poco por leer, creo que todos podemos estar convecidos de algo: los algoritmos genéticos funcionan y lo hacen bien.

Los algoritmos genéticos pueden verse como un método de búsqueda robusto, lo que implica que el problema que se desea resolver puede tener ligeros o grandes cambios, tan ligeros como el cambio de una variable o

tan grandes como el cambio de un problema por otro. No importa, el algoritmo genético seguirá funcionando.

La robustez de los algoritmos genéticos ha sido tomada de una simulación de los métodos genéticos presentados en la naturaleza, que nos ha aleccionado por miles de años, acerca de lo que significa resistir a pequeños o grandes cambios través de las poblaciones de seres que la habitamos.

El éxito de los algoritmos genéticos radica sus procesos de la reproducción, muerte, selección y fenómenos cromosómicos. La reproducción permite la búsqueda de nuevas soluciones al entorno al que se enfrenta la población.

La muerte tiene la función de eliminar aquellos seres de la población que han probado no ser eficientes frente al medio, y no puede verse como una acción cruel, sino más bien como una forma de fortalecer a la población ante su ambiente.

La selección no es concebible como un fenómeno determinístico, puesto que no siempre muere el más débil, ni solo se reproduce el más fuerte. La selección es en realidad un fenómeno probabilístico que sirve de sustento para la realización tanto de la reproducción como de la muerte. Con la selección probabilística de los más fuertes para su reproducción, facilitamos la búsqueda de una nueva generación que aporte mejores resultados al entorno que enfrenta la población. Con la selección probabilística de los más débiles para su muerte provocamos que se fortalezca la población.

Los fenómenos cromosómicos, tales como los aquí revisados, entrecruzamiento, mutación e inversión, han probado ser un método eficiente que permite el diseño aleatorio de nuevas formas de solución y que pueden resultar ser excelentes formas de resolver el ambiente de la población. También existe el riesgo, de que algún fenómeno cromosómico por sus propiedades aleatorias, pueda crear seres que resulten verdaderos "monstruos", que lejos de ayudar, decrementan la fuerza promedio poblacional. Sin embargo, los procesos selección, reproducción y muerte están encargados de la conservación y propagación de los habitantes más exitosos, así como también de la extinción de tales "monstruos".

Cada vez, será menos extraño en el ámbito informático, el encontrar como opción de solución a los algoritmos genéticos. Muchas universidades de reconocido prestigio han incluido ya, en sus mapas curriculares de nivel de licenciatura y posgrado de informática, ciencias computacionales, inteligencia artificial y similares, una materia dedicada al estudio de algoritmos genéticos. Los algoritmos genéticos se convertirán no solo un tema de interés para la informática, sino a materias como la biología, investigación de operaciones, ingeniería, administración, finanzas, contaduría, en fin, a cualquier parte donde se tenga un problema que requiera un proceso de búsqueda de soluciones confiable, robusto y eficiente, pues los algoritmos genéticos no restringen su área de aplicación a la informática, sino a todos aquellos campos en los que la informática se puede aplicar.

BIBLIOGRAFÍA

Bibliografía básica acerca de algoritmos genéticos

- Goldsberg, David;
  Genetic Algorithms in Search, Optimization, and Machine Learning;
  Addison-Wesley; U.S.A, 1989.
- Lille Borja, José de;
  Biología;
  Porrua; México, 1959.
- Shao, Stephen P.;
  Estadística para economistas y adminstradores de empresas;
  Herrero hnos.; México, 1988 (20ª edición).

Bibliografía básica acerca del lenguaje C++

- Eckel Bruce;
  Using C++;
  McGraw Hill; U.S.A., 1991.
- Hekmatpour, Sharam;
  C++ a book and disk guide for C Programers;
  Prentice Hall; U.S.A., 1992.

- Narkakati, Naba;
  The Waite Group's Turbo C++ Bible;
  SAMS; U.S.A., 1990.

Bibliografía suplementaria acerca de algoritmos genéticos

- Booker, L. B.;
  Improving search in genetic algorithms;
  Pitman; London, 1987.
- Davis, L.;
  Genetic algorithms and simulated annealing;
  Pitnam; London, 1987.
- Davis, L. & Combs S.;
  Optimizing network link sizes with genetic algorithms;
  North-Holland; Amsterdam, 1990.
- Davis, L. & Steenstrup, M.;
  Genetic Algorithms and simulated annealing. An overview;
  Pitnam; London, 1987.
- Fogel, L. J., Owens, A. J. & Walsh, M. J. ;
  Artificial intelligence through simulated evolution;
  John Wiley; New York, 1966.
- Goldberg, D. E.;
  The genetic algorithm aproach: Why, how, and what next?;
  Plenum Press; New York, 1986.
- Goldberg, D. E.;
  Simple genetic algorithms and the minimal, deceptive problem;
  Pitnam; London, 1987

- Goldberg, D. E.;
  Genetics-based machine learning: Whence it came;
  North-Holland; Amsterdam, 1990.
- Holland, J. H.;
  Adaptation in natural and artificial systems
  The University of Michigan Press; Ann Arbor, 1975

Bibliografía suplementaria acerca del Lenguage C++

- Augie Hansen;
  Proficient C;
  Microsoft Press; U.S.A., 1987.
- Hancock, Les & Kringer, Morris;
  The C primer;
  McGraw Hill; U.S.A., 1985 (2ª edición).
- Kernigan, Brian & Ritchie, Dennis M.;
  The C Programming Language;
  Prentice Hall; U.S.A., 1978.
- Kochan, Stephen G.;
  Programming in C;
  Hayden; U.S.A., 1983.
- La Fore, Robert;
  C Programming using Turbo C++;
  SAMS; U.S.A., 1990.
- La Fore, Robert;
  Turbo C Programming for the IBM;
  Howard W. Sams & Company; U.S.A., 1987.

- Prata, Stephen G. & Martin, Donald;
  C Primer Plus;
  SAMS; U.S.A., 1984.
- Schustack, Steve;
  Variations in C;
  Microsoft Press; U.S.A., 1985.
- Shildt, Herbert;
  C: Power user's guide;
  McGraw Hill; U.S.A., 1990.
- Shildt, Herbert;
  Artificial intelligence usign C;
  McGraw Hill; U.S.A., 1988.