



Universidad de Murcia

**Material didáctico para la asignatura
Sistemas Inteligentes
de 3º de Grado en Informática**

AUTORES:

- José Manuel Cadenas Figueredo
- María del Carmen Garrido Carrera
- Raquel Martínez España
- Santiago Paredes Moreno

Preámbulo

El objetivo de este material es servir como una guía de apoyo al alumno en el desarrollo e implementación de Algoritmos Genéticos para la resolución de problemas en el marco de las prácticas de la asignatura **Sistemas Inteligentes de 3º (SSII-3º) del Grado en Ingeniería Informática**.

Este documento incluye una introducción al lenguaje de programación orientado a objetos “C++” con todos aquellos elementos que el alumno necesitará para llevar a cabo la práctica y una descripción de aquellos elementos que se necesita conocer de la “**librería C++ GAlib**” [1] (librería de funciones en C++ que proporciona al programador de aplicaciones un conjunto de objetos para el desarrollo de Algoritmos Genéticos). Se utilizan tres ejemplos didácticos seleccionados exclusivamente para facilitar al alumno el trabajo con dicha librería.

Pretendemos con el desarrollo de este material que el alumno dedique su principal esfuerzo en el análisis de los problemas planteados en el ámbito de los SSII-3º y en el diseño de una solución a los mismos, y no en las herramientas para su implementación, dado que no se trata de una asignatura de Programación.

Índice General

I	INTRODUCCIÓN AL LENGUAJE C++	1
1	Variables, funciones y operadores	3
1.1	Introducción	3
1.2	Ámbito de las variables	4
1.3	Casting	5
1.4	Punteros	7
1.5	Funciones: paso por valor y paso por referencia	11
2	Definición de clases	15
2.1	Introducción	15
2.2	Definición de clases	15
2.3	Constructores	17
2.4	Destructores	21
2.5	Ejemplo	24
3	Herencia en C++	27
3.1	Introducción	27
3.2	Herencia simple y visibilidad en la herencia	28
3.3	Herencia múltiple	37
4	Entrada/Salida mediante Ficheros	45
4.1	Introducción	45
4.2	Manejo de ficheros	47
4.2.1	Ejemplo1: Escribiendo en un fichero	50
4.2.2	Ejemplo 1.1: Escribiendo en un fichero utilizando otro constructor	50

4.2.3	Ejemplo 2: Leyendo de un fichero línea a línea	51
4.2.4	Ejemplo 3: Leyendo un fichero y copiándolo en otro, utilizando las funciones <code>write</code> y <code>getline</code>	52
4.2.5	Ejemplo 3.1: Leyendo un fichero y copiándolo en otro, utilizando los operadores de dirección	52
II	LIBRERÍA GALib	55
5	Introducción a la librería GALib	57
5.1	Introducción	57
5.2	Creación de un proyecto que use GALib	58
5.2.1	Instalación de la librería GALib en el entorno Dev-C++	58
5.2.2	Esquema básico de un algoritmo genético utilizando GALib	60
5.3	Descripción de los problemas utilizados	62
5.3.1	Optimización de una función	63
5.3.2	Problema de las N reinas	63
5.3.3	Problema de la mochila	63
5.4	Generadores aleatorios y semillas	64
6	Clases GAGenome	67
6.1	Introducción	67
6.2	Métodos principales	68
6.3	Clase GA1DBinaryStringGenome	70
6.3.1	Constructor de clase	70
6.3.2	Operadores genéticos por defecto y disponibles	70
6.3.3	Representación para el problema de minimizar una función con restricciones	72
6.4	Clase GA1DArrayGenome<T>	74
6.4.1	Constructor de clase	74
6.4.2	Operadores genéticos por defecto y disponibles	74
6.5	Clase GA1DArrayAlleleGenome<T>	76
6.5.1	Constructor de clase	76
6.5.2	Definición de los alelos: Clase GAAlleleSet<T>	76

6.5.3	Operadores genéticos por defecto y disponibles	77
6.5.4	Representación para el problema de las N reinas	77
6.6	Función Objetivo/Fitness	78
6.6.1	Ejemplos de funciones objetivo	79
7	Definición del algoritmo genético	83
7.1	Introducción	83
7.2	La clase GASimpleGA	84
7.2.1	Constructor de clase	85
7.2.2	Métodos principales	85
7.2.3	Terminación del algoritmo genético	87
7.3	Impresión de valores de la ejecución	88
7.3.1	Objeto GAStatistics	88
7.3.2	Métodos principales	89
7.4	Código completo para la optimización de una función	90
7.5	Código completo del ejemplo de las N reinas	93
7.6	Resolviendo el problema de la mochila 0-1	97
III	BIBLIOGRAFÍA	105

Parte I

**INTRODUCCIÓN AL
LENGUAJE C++**

Capítulo 1

VARIABLES, FUNCIONES Y OPERADORES

1.1 Introducción

Las variables en un programa se definen como entidades cuyos valores pueden ir variando a lo largo de la ejecución. El lenguaje C++ [2, 3, 6] utiliza los mismos tipos de variables que el lenguaje C, es decir, `void`, `char`, `int`, `float` y `double`, además del tipo `enum` para definir enumerados y del tipo `bool` para tratar con booleanos. Hay que tener en cuenta que existen ciertos modificadores que modifican la longitud de los tipos básicos. Estos modificadores son: `short`, `signed`, `unsigned` y `long`.

La sintaxis para definir una variable es primero definir el tipo y después definir el nombre de la variable. Un ejemplo sería:

```
1 [signed | unsigned] int posicion;
```

Como se puede apreciar en el ejemplo, al definir el tipo también le podemos aplicar un modificador. En el caso del tipo entero, el modificador aplicado es para considerar si el tipo entero es con signo o sin signo. Para utilizar estos modificadores hay que tener en cuenta el uso que se le vaya a dar la variable, ya que si una variable es declarada como un entero sin signo, el programa no dará los resultados esperados si ésta se utiliza para realizar ciertas operaciones matemáticas.

Además de los modificadores, hay que tener en cuenta algunas reglas a la hora de poner el nombre de una variable en el lenguaje C++. Las dos reglas más básicas a considerar son:

1. El nombre de una variable puede estar formado por letras (mayúsculas o minúsculas), números y caracteres no alfanuméricos como el `_`, pero nunca pondrá contener una coma, un punto y coma, un guión, un punto, símbolos matemáticos o interrogaciones. El nombre de una variable nunca puede empezar por número.
2. El lenguaje C++ distingue entre mayúsculas y minúsculas, por lo que el nombre de variable `Mivariable` es diferente a la variable llamada `mivariable`.

1.2 Ámbito de las variables

El lenguaje C++ permite definir variables globales y variables locales. Una variable es local cuando se define en el interior de una función. Esta variable será accesible desde dentro de la función y tendrá una duración temporal en tiempo de ejecución hasta que la función acabe. Una vez que la función finaliza la variable es destruida. En otras palabras, el ámbito de acceso de una variable declarada dentro de una función es un ámbito local y el ámbito temporal de la misma es el tiempo en el cuál se está ejecutando tal función. Hay que tener en cuenta que una variable local será visible y estará accesible para el código de la función programada después de la declaración de tal variable. Además, las variables declaradas dentro de bucles o de sentencias condicionales, tienen visibilidad sólo dentro de tales sentencias.

Por otra parte, las variables globales tienen ámbito global tanto en tiempo como en acceso, ya que una variable global durará durante toda la ejecución del programa y podrá ser accedida desde las diferentes funciones del programa. Aunque, hay que añadir que una variable global será accesible por aquellas funciones declaradas posteriormente en orden de código. Por último debemos de tener en cuenta que no se debe de abusar del uso de variables globales en los programas, ya que en términos de seguridad, las variables globales son consideradas poco seguras, puesto que su accesibilidad puede comprometer en ciertas circunstancias el valor de las mismas.

Un aspecto a tener en cuenta es que los parámetros de una función también pueden ser considerados como variables locales, dependiendo de si estos parámetros fueron pasados por valor o por referencia, pero este tema lo vamos a tratar un poco más adelante.

Un caso a tener en cuenta es qué ocurre cuando nos encontramos dentro de una función y declaramos una variable con el mismo nombre que una variable global. Por

ejemplo:

```
1 int a;  
2 int main() {  
3     int a;  
4     a = 5;  
5 }
```

En este ejemplo tenemos la variable `a` declarada de forma global y dentro de la función `main` tenemos otra variable declarada con el mismo nombre. La pregunta que nos hacemos es, ¿la asignación del valor 5 se lo estamos realizando a la variable local o a la variable global? La respuesta a esta pregunta es clara, la asignación se realiza sobre la variable local, ya que es el ámbito de acceso que podríamos llamar más cercano. Si quisiéramos realizar esta asignación sobre la variable global, debemos de utilizar el operador de ámbito `::`. Poniendo delante de la variable dicho operador lo que conseguimos es cambiar el ámbito y acceder al ámbito global. Esta aplicación es una de las muchas que tiene dicho operador. Volviendo al ejemplo, tendríamos:

```
1 int a;  
2 int main() {  
3     int a; // Variable local que enmascara a la global  
4     a = 5; // Asignación a la variable local  
5     ::a = 3; // Asignación a la variable global  
6     return 0;  
7 }
```

Como resumen podemos indicar que los parámetros de una función se comportan del mismo modo que las variables locales y tienen ámbito local a la función en el sentido de ámbito de acceso. Además las variables declaradas dentro de las sentencias condicionales y los bucles sólo tienen visibilidad dentro de tales ámbitos. Por último, recordar que una variable será accesible y visible para el código posterior a la declaración de tal variable.

1.3 Casting

Cuando se realizan operaciones matemáticas en el lenguaje C++, los operandos que intervienen pueden ser de distinto tipo de datos, sin embargo, los datos se convierten de forma temporal al operando que tenga una mayor precisión. Estos casting se llevarán

a cabo siempre y cuando no se produzca pérdida de información. Cuando esto ocurra será el compilador el encargado de avisar de ello. Este tipo de casting se realiza de forma implícita, pero además en las operaciones matemáticas, cuando se realiza una asignación entre diferentes tipos de datos, también podemos llevar a cabo un casting indicando entre paréntesis la conversión del nuevo tipo. En este caso puede ocurrir:

- Si se asigna una variable de tipo `float` o `double` a una de tipo `int`, el valor de la variable pierde la parte decimal, asignándose solamente la parte entera.
- Si se asigna una variable de tipo `double` a una de tipo `float` el valor de la variable queda redondeado, ya que pierde parte de la precisión en los decimales.
- Si una variable de tipo `int` se asigna a una variable también de tipo `int` pero de menor precisión o de tipo `char`, algunos de los bits más significativos pueden perderse.

Un ejemplo sería:

```
1 float a = 2.3015;
2 int aParteEntera = (int) a;
```

Para realizar este tipo de casting, el lenguaje C++ dispone, además del uso de los paréntesis (cuyo uso no es recomendado), del operador `static_cast`. Este operador es el primer tipo de conversión al que se suele recurrir para realizar conversiones implícitas entre tipos, por ejemplo de `int` a `float` o de un puntero de cierto tipo a un puntero `void` (este tipo de conversión también está permitida en C++). La sintaxis de este operador es:

```
1 static_cast<tipo>("expresión");
```

Este operador realiza la conversión de tipos durante la compilación del programa, de modo que no crea más código durante la ejecución, descargando esa tarea en el compilador. Además de los tipos de conversiones comentados, el operador `static_cast` puede realizar conversiones entre punteros de clases relacionadas, es decir, conversiones de una clase derivada a su clase base y viceversa. Los conceptos de clase base y clase derivada serán comentados más adelante en el capítulo de herencia. Este tipo de casting

no realiza ninguna comprobación de seguridad en tiempo de ejecución, por lo tanto, esta tarea queda pendiente para el programador. Un ejemplo de este tipo de operador sería:

```
1 class Base1 { ... };
2 class Derivada1: public Base1 { ... };
3 Base1 * b1 = new Base1;
4 Derivada1 * d1 = static_cast<Derivada1 *>(b1);
```

Sin embargo, hay que tener en cuenta que este código podría dar errores en tiempo de ejecución si el objeto `d1` fuera desreferenciado.

Un ejemplo más simple donde se produce una conversión de un tipo `float` a un tipo `int` sería:

```
1 float decimal = 1.2563;
2 int entera = static_cast<int>(decimal);
```

1.4 Punteros

Un puntero es una variable que almacena direcciones de memoria y que se define con el operador `*`. Como puede almacenar una dirección de memoria de una variable de cualquier tipo, su definición es, “tipo de la variable” “operador de puntero” “nombre de la variable”. Un ejemplo sería:

```
1 char * var1;
```

La variable `var1` es de tipo puntero a `char`, es decir, esta variable apunta a la primera dirección de memoria donde se va a almacenar un valor `char`. El operador de puntero permite realizar el paso por referencia de valores, pero este concepto lo veremos un poco más adelante. El operador de puntero `*`, también llamado operador de indirección se utiliza para obtener y/o modificar el valor de la variable a la que apunta. Por otro lado, tenemos el operador de dirección `&`, que devuelve la dirección de memoria donde comienza el valor de la variable. Un ejemplo de uso de estos operadores es:

```
1 int var1 = 5;
2 int * pvar1;
3 pvar1 = &var1;
```

Creamos una variable de tipo `int` inicializada a 5. Luego creamos un puntero a `int` llamado `pvar1` y por último el puntero que hemos creado pasa a apuntar a la variable `var1`, por lo que `pvar1` es un puntero a la variable entera `var1`.

Los punteros facilitan y permiten trabajar de forma aritmética con los vectores. Un vector que no tiene índice puede definirse como un puntero a la primera posición del vector. Por ejemplo, podemos definir las siguientes variables:

```
1 double vec1 [10];  
2 double * pvec1;  
3 pvec1 = &vec1;  
4 pvec1 = &vec1 [0];
```

Las dos últimas instrucciones realizan la misma función, consiguen que el puntero `pvec1` apunte a la primera posición del vector `vec1`. Siguiendo con el ejemplo, vamos a ver cómo la aritmética nos permite avanzar por un vector sin utilizar índices. Así tenemos que:

```
1 double v = *(pvec1 + 4);
```

Con esta operación lo que se realiza es que la variable `v` contenga el valor del elemento quinto del vector. Al sumarle 4 al puntero lo que se le está indicando es que se coloque apuntando en la posición en la que estaba más cuatro, por lo que como estaba en la posición 0 pasa a apuntar a la posición 4. Un aspecto a comentar son los paréntesis. El operador puntero `*` queda fuera del paréntesis porque tiene una prioridad mayor que el operador de suma. Si no indicáramos los paréntesis, se le sumaría 4 al valor contenido en la posición 0 del vector que es a la que está apuntando actualmente el puntero `pvec1`. Al igual que hemos utilizado el operador de suma se puede utilizar el operador de resta para moverse a posiciones anteriores, siempre teniendo en cuenta las dimensiones del vector, ya que si no se tienen en cuenta se producirán errores en tiempo de ejecución ya que en tiempo de compilación no se realiza ninguna comprobación de este tipo.

Un aspecto a comentar es qué ocurre con los punteros y las constantes. Como regla general podemos indicar que un puntero es constante cuando siempre apunta a una misma dirección de memoria, sin embargo si la variable no está definida como constante el valor puede ser modificado. Para que el valor no pueda ser modificado tiene que ser constante tanto el puntero como la variable. Por ejemplo:


```
1 #include <iostream>
2 using namespace std;
3
4 int main (){
5     int var = 1;
6     const int * const pvar = &var;
7
8     cout << *pvar << endl;
9
10    //Esta instrucción provoca un error ya que no se puede alterar el valor
11
12    *pvar = 5;
13 }
```

El operador puntero también permite definir una matriz de dos dimensiones que se crea dinámicamente. Un ejemplo sería:

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5
6     int **matriz;
7     int * matriz1[10];
8
9     matriz = new int*[10];
10
11    for(int i = 0; i < 10; i++){
12        matriz[i] = new int[10];
13        matriz1[i] = new int[10];
14    }
15
16    for(int i = 0; i < 10; i++)
17        for(int j = 0; j < 10; j++){
18            matriz[i][j]=i;
19            matriz1[i][j]=i;
20        }
21
22    for(int i = 0; i < 10; i++)
23        for(int j = 0; j < 10; j++)
24            cout << "matriz[" << i << ", " << j << "]= " << matriz[i][j]
                << endl;
```

```
25
26     for(int i = 0; i < 10; i++)
27         for(int j = 0; j < 10; j++)
28             cout << "matriz1[" << i << ", " << j << "]=" << matriz1[i][j]
29                 << endl;
30
31     system("pause");
32     return 0;
33 }
```

Como último aspecto a comentar de los punteros, vamos a destacar los punteros a funciones. Hay que tener en cuenta que un puntero no es más de una dirección de memoria donde se produce el comienzo de algún tipo de instrucción. Los punteros a funciones apuntan a la dirección de memoria donde comienza el código de una función. La ventaja de estos punteros a funciones es que se pueden pasar como parámetro y se puede retornar en las funciones. La sintaxis para este tipo de punteros es:

```
1 "Tipo" (*"nombreFunción") ("parámetros");
```

La llamada a una función como un puntero se realiza de la misma forma que si se llamara a una función normal, puesto que el puntero es un alias de la función. Un ejemplo de punteros a funciones es:

```
1 #include <iostream>
2 using namespace std;
3
4 int calcula1(int, int (*)(int));
5 int calcula(int);
6
7 int main(){
8     int arg1 = 3;
9
10    if ((calcula1(arg1, calcula))>0)
11        cout << "El calculo no es exacto" << endl;
12    else
13        cout << "El calculo es exacto" << endl;
14
15    system("pause");
16    return 0;
17 }
```

```
18
19 int calcula1(int arg1, int (* cal)(int )){
20     return (int)(*cal)(arg1) % arg1;
21 }
22 int calcula(int arg){
23     return (int)arg * 7;
24 }
```

En el ejemplo se puede observar la definición de dos funciones. La primera de ellas tiene dos parámetros, donde el segundo de ellos es un puntero a una función que devuelve un entero y recibe un único parámetro entero. Para usar este parámetro se hace una llamada a la función utilizando el puntero y pasándole el parámetro correspondiente.

1.5 Funciones: paso por valor y paso por referencia

Anteriormente hemos comentado que los parámetros de una función tienen las mismas características que las variables locales de una función siempre y cuando el paso de tales parámetros fuera por valor. Pero, ¿qué significa pasar parámetros por valor o por referencia? Hasta el momento, a la hora de llamar a una función siempre hemos pasado los parámetros de la misma forma. Un ejemplo sería:

```
1 #include <iostream>
2 using namespace std;
3
4 int suma(int a, int b);
5
6 int main(){
7     int a, b;
8     a = 10;
9     b = 20;
10    int resul = suma(a, b);
11    cout << "La suma de " << a << " y " << b << " es: " << resul << endl;
12
13    system("pause");
14    return 0;
15 }
16
17 int suma(int a, int b) {
18     int r = a + b;
19     a = 5;
```

```
20     b = 5;
21     return r;
22 }
```

La salida por pantalla de la ejecución de este programa es:

La suma de 10 y 20 es: 30

Como vemos a pesar de que la función `suma` modifica los parámetros `a` y `b` después de sumarlos y que se llama a la función `suma` antes de imprimir, el resultado de `a` y `b` no se modifica sino que se quedan los valores que tenían las variables `a` y `b` en la función principal. Así es como hasta el momento hemos estado definiendo la llamada a las funciones. A este caso se le llama pasar parámetros por valor a una función. Cuando se pasa parámetros a una función por valor, como es el caso de los parámetros de la función `suma`, significa que la función que recibe los parámetros crea una “copia” de dichos parámetros y aunque los parámetros sean modificados dentro de la función, al terminar la función esta “copia” de los parámetros se elimina, por lo tanto, al volver a la función inicial (en el ejemplo la función `main`), las variables pasadas como parámetro (`a` y `b` en el ejemplo) siguen teniendo el mismo valor.

Sin embargo, el caso contrario es pasar los parámetros por referencia. Cuando los parámetros de una función en el lenguaje C++ se quieren pasar por referencia hay que indicarlo de forma explícita. Antes de introducirnos en este tema, vamos a ver para qué sirve una referencia.

Una referencia en el lenguaje C++ no es más que un alias para un objeto. Si definimos una referencia a un objeto, este objeto podrá ser modificado tanto si modificamos el valor del objeto directamente como si modificamos la referencia. Las referencias son una forma de tener diferentes identificadores de un objeto sin necesidad de utilizar punteros. Para definir una referencia utilizamos el operado de referencia “&”. Un ejemplo sería:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int var1;
6     int &ref1 = var1;
7     int &ref2 = var1;
8
9     var1 = 5;
```

```
10 cout << ref2 << endl;
11
12 system("pause");
13 return 0;
14 }
```

Como vemos en el ejemplo, la variable `var1` tiene dos referencias que son las variables `ref1` y `ref2`. Al asignar un valor a `var1` e imprimir una de sus referencias, se obtiene el mismo valor que si imprimiéramos la variable en sí.

Una vez definido qué es una referencia, volvemos al caso de pasar parámetros a una función por referencia. Cuando se le pasan parámetros a una función por referencia, como ya hemos comentado, una referencia no es más que un alias, por lo tanto si desde una función `A` se llama a una función `B`, donde la función `B` recibe los parámetros `x` e `y` por referencia, tenemos que el valor de `x` e `y` en la función `B` pueden ser modificados y tales modificaciones serán también visibles desde la función `A`. Para definir que una función recibe los parámetros por referencia, debe de indicarse un “&” delante de cada parámetro que quiera ser recibido por referencia. Veamos un ejemplo:

```
1 #include <iostream>
2 using namespace std;
3
4 int funcionA(int &x, int &y);
5 int funcionB(int a, int &b);
6
7 int main(){
8     int v1, v2;
9     v1 = 1; v2 = 4;
10    cout << "v1 y v2 = " << v1 << " y " << v2 << endl;
11    funcionA(v1, v2);
12    cout << "Tras llamar a la funcionA - v1 y v2: " << v1 << " y " << v2 <<
    endl;
13    funcionB(v1, v2);
14    cout << "Tras llamar a la funcionB - v1 y v2: " << v1 << " y " << v2 <<
    endl;
15
16    // Lo comentado a continuación no es posible hacerlo, ya que no se
    // pueden pasar constantes como parámetros
17    // cout << "FunciónB(5,2) - " << funcionB(5,2) << endl;
18
19    system("pause");
```

```
20     return 0;
21 }
22
23 int funcionA(int &x, int &y){
24     x = x + 55;
25     y = y + 55;
26     return y + y;
27 }
28
29 int funcionB(int a, int &b){
30     a = a + 1;
31     b = b - 1;
32     return a * a;
33 }
```

La salida del programa es:

v1 y v2 = 1 y 4

Tras llamar a la funcionA - v1 y v2: 56 y 59

Tras llamar a la funcionB - v1 y v2: 56 y 58

Como vemos, los valores de las variables v1 y v2 se modifican cuando se llama a la funcionA, sin embargo, al llamar a la funcionB sólo el valor de v2 se ve afectado, ya que la funcionB sólo recibe un parámetro por referencia, el otro parámetro es recibido por valor.

Capítulo 2

Definición de clases

2.1 Introducción

El lenguaje C++ tiene la capacidad de permitir realizar una programación estructurada y también nos proporciona herramientas para realizar una programación orientada a objetos. Una de las herramientas clave que nos proporciona C++ orientado a objetos son las clases. Una clase no es más que una especificación a seguir para construir objetos. Un objeto lo podemos definir como una entidad autónoma que tiene una funcionalidad bien definida. Haciendo una analogía con el lenguaje de programación C, podemos ver una clase como una estructura en C, teniendo en cuenta que las clases de C++ nos proporcionan ciertas ventajas, sencillas pero bastante potentes.

2.2 Definición de clases

Continuando con la analogía del lenguaje C, para definir una clase lo que debemos de hacer es cambiar la palabra reservada `struct` por `class`. Un ejemplo sencillo para empezar a ver la definición de las clases, sería el siguiente:

```
1 class Editorial{
2     char * nombre;
3     char * direccion;
4     int identificador;
5     int nPublicaciones;
6     void informacion();
7     void addPublicacion(char * titulo);
8 };
```

Este código define la clase `Editorial` que está compuesta de los atributos `nombre`, `direccion`, `identificador` y número de publicaciones (`nPublicaciones`). Además, en la definición de la clase se indican las cabeceras de los métodos de la clase.

Una de las características de la definición de clases es que se puede especificar dentro de una clase diferentes especificaciones de acceso. Una especificación de acceso indica cuál es el nivel de visibilidad de los atributos o métodos que hay definidos. Las especificaciones de acceso que se pueden indicar son `private`, `public` y `protected`.

La especificación de acceso `private` indica que los atributos o funciones definidos en tal especificación sólo son accesibles desde dentro de la clase, es decir, que ninguna función externa a la clase o función de clase derivada tendrá acceso. La especificación `public` indica que los atributos y funciones definidos en esta especificación serán accesibles desde cualquier parte donde el objeto sea accesible. Por último la especificación de acceso `protected`, se refiere a que los atributos y funciones de la clase se comportarán como de acceso `public` para las clases derivadas y se comportarán como `private` para las funciones externas. Para definir la especificación de acceso simplemente hay que indicar la palabra reservada seguida de “:”. Cuando no se especifica nada en una clase, por defecto se considera que la especificación de acceso es `private`.

Un ejemplo de cómo definir las especificaciones de acceso es el siguiente:

```
1 class Editorial{
2
3 private:
4     char * nombre;
5     char * direccion;
6     int identificador;
7     int nPublicaciones;
8
9 public:
10    void informacion();
11    void addPublicacion(char * titulo);
12
13};
```

Un aspecto a comentar es que las funciones las podemos implementar dentro o fuera de la clase. En el código mostrado arriba solamente se están declarando las funciones y atributos, por lo que las funciones van a ser implementadas fuera de la clase. Para

identificar que se están implementando las funciones de una clase, debemos de utilizar el operador de ámbito `::` indicando antes de tal operador a qué clase pertenecen. Así por ejemplo para implementar la función `informacion` fuera de la clase deberíamos de implementarla de la siguiente manera:

```
1 void Editorial::informacion() {  
2     cout << "El nombre de la editorial es " << nombre << endl;  
3 }
```

Un aspecto a tener en cuenta es que no se reserva memoria para un objeto de una clase hasta que este objeto es creado. Es el constructor de la clase el encargado de crear el objeto y de reservar la memoria necesaria para inicializar aquellos campos que sean requeridos. De la misma forma, cuando un objeto ya no es necesario utilizarlo hay que llamar al destructor para liberar aquella memoria que se reservó en el constructor de tal objeto. Vamos a ver de forma más detallada algunos aspectos acerca de los constructores y destructores de una clase.

2.3 Constructores

Los constructores, como hemos comentado, son funciones que nos sirven para crear el objeto e inicializar los valores pertinentes reservando memoria en caso de que sea necesario, es decir, mediante la llamada a un constructor inicializamos los valores de un objeto al mismo tiempo que creamos el objeto. Los constructores son funciones especiales debido a las siguientes razones:

- No tienen tipo de retorno, ni retornan ningún valor.
- No se heredan.
- El nombre de la función debe de ser exactamente el mismo que el de la clase.
- Un constructor siempre debe de ser una función pública, ya que es la que crea el objeto. No tiene sentido un constructor `private` o `protected`, ya que un constructor, la mayoría de las veces, será llamado desde funciones exteriores de la clase y al no poder heredarse, no tiene sentido que sea público para las clases derivadas y no para el resto de funciones.

Si en una clase no se define un constructor, el compilador crea uno por defecto sin parámetros para crear el objeto, pero este constructor no inicializa absolutamente nada, por lo que los atributos que deban de ser inicializados, contendrán una información incorrecta. Un aspecto importante es que si se ha definido un constructor con argumentos, cuando se llame a ese constructor habrá que pasarle los argumentos pertinentes de forma obligatoria. Siguiendo con el ejemplo que teníamos anteriormente de la editorial, veamos cómo sería el constructor de tal clase:

```
1 class Editorial{
2
3 private :
4     char * nombre;
5     char * direccion;
6     int identificador;
7     int nPublicaciones;
8
9 public :
10    Editorial(char * name, char * address , int id , int npubli);
11    void informacion();
12    void addPublicacion(char * titulo);
13 };
14
15 Editorial::Editorial (char * name, char * address , int id , int npubli){
16     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
17     strcpy(nombre,name);
18     direccion = new char [strlen(address)+1];
19     strcpy(direccion ,address);
20     identificador = id;
21     nPublicaciones = npubli;
22 }
```

Como podemos observar dentro de la clase hemos declarado el constructor y fuera de la clase lo hemos implementado indicando mediante el operador de ámbito `::` que pertenece a la clase `Editorial`. Dentro de la implementación del constructor hemos reservado la memoria correspondiente para los atributos `nombre`, `direccion` y hemos inicializado los valores.

Se pueden definir tantos constructores como sea necesario, es decir, el lenguaje C++ permite la sobrecarga de constructores, así si hay circunstancias donde no es necesario inicializar atributos cuando se crea el objeto, pero hay otras situaciones donde sí,

definimos los constructores modificando el número de parámetros que reciben. Continuando con el ejemplo de la editorial, vamos a sobrecargar el constructor creando otro constructor más con diferentes parámetros:

```
1 class Editorial{
2
3 private:
4     char * nombre;
5     char * direccion;
6     int identificador;
7     int nPublicaciones;
8
9 public:
10    Editorial(char * name, char * address, int id, int npubli);
11    Editorial(char * name, char * address, int id);
12    void informacion();
13    void addPublicacion(char * titulo);
14 };
15
16 Editorial::Editorial (char * name, char * address, int id, int npubli){
17     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
18     strcpy(nombre,name);
19     direccion = new char[strlen(address)+1];
20     strcpy(direccion ,address);
21     identificador = id;
22     nPublicaciones = npubli;
23 }
24
25 Editorial::Editorial(char * name, char * address, int id){
26     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
27     strcpy(nombre,name);
28     direccion = new char[strlen(address)+1];
29     strcpy(direccion ,address);
30     identificador = id;
31     nPublicaciones = 0;
32 }
```

En el ejemplo vemos cómo hemos definido dentro de la clase dos constructores, con diferentes parámetros y después fuera de la clase los hemos implementado al igual que hemos realizado anteriormente utilizando el operador de ámbito `::`.

Un constructor importante es el llamado constructor de copia. Este constructor es utilizado para, como su nombre indica, crear una copia del objeto que recibe como parámetro. Este tipo de constructor sólo recibe un parámetro que es una referencia al objeto de la misma clase. Siguiendo con el ejemplo de la clase editorial, veamos cómo funciona el constructor de copia.

```
1 class Editorial{
2
3 private:
4     char * nombre;
5     char * direccion;
6     int identificador;
7     int nPublicaciones;
8
9 public:
10    Editorial(char * name, char * address, int id, int npubli);
11    Editorial(char * name, char * address, int id);
12    Editorial(const Editorial &e);
13    void informacion();
14    void addPublicacion(char * titulo);
15 };
16
17 Editorial::Editorial (char * name, char * address, int id, int npubli){
18     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
19     strcpy(nombre,name);
20     direccion = new char[strlen(address)+1];
21     strcpy(direccion ,address);
22     identificador = id;
23     nPublicaciones = npubli;
24 }
25
26 Editorial::Editorial(char * name, char * address, int id){
27     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
28     strcpy(nombre,name);
29     direccion = new char[strlen(address)+1];
30     strcpy(direccion ,address);
31     identificador = id;
32     nPublicaciones = 0;
33 }
34
35 Editorial::Editorial(const Editorial &e){
36     nombre = (char *) malloc((strlen(e.nombre)+1) * sizeof(char));
```

```
37     direccion = new char [ strlen ( e . direccion ) + 1 ];
38     strcpy ( nombre , e . nombre );
39     strcpy ( direccion , e . direccion );
40     identificador = e . identificador ;
41     nPublicaciones = e . nPublicaciones ;
42 }
```

Como se puede observar el constructor de copia lo que hace es hacer una copia en profundidad del objeto que se recibe como parámetro. Como se aprecia no se hace una asignación de punteros sino que se copia el contenido de la memoria explícitamente con `strcpy`. Esto es importante porque si se hace una copia de puntero como tal, al modificar el nuevo objeto estaríamos modificando también el objeto que se ha pasado para copiar. Es importante destacar que durante el ejemplo que hemos detallado se han especificado dos formas de reservar memoria en C++, utilizando la función `malloc`, heredado del lenguaje C y creando un objeto tipo `char *` con el operador `new` e indicándole la cantidad de espacio en memoria a reservar.

El constructor de copia es llamado no sólo cuando se quiere crear una copia de un objeto sino que el compilador en ciertas situaciones también lo utiliza. Una de esas situaciones es cuando se iguala un objeto a otro a la misma vez que se inicializa. Por ejemplo:

```
1 Editorial e1 ("Thomson", "Avenida de Madrid", 3, 25634);
2 Editorial e2 = e1;
```

En el ejemplo hemos declarado el objeto `e1` de la clase `Editorial` y hemos llamado al constructor con sus parámetros y luego hemos declarado otro objeto `e2` de tipo `Editorial` y lo hemos inicializado igualándolo al objeto `e1`, por lo que internamente el compilador al analizar esta sentencia lo que hace es llamar al constructor de copia.

2.4 Destruidores

Al igual que los constructores, los destructores son funciones especiales en el lenguaje C++. El destructor es una función que se llama bien directamente o bien de forma automática al querer eliminar un objeto, es decir, desde una función se puede destruir un objeto que ya no sea necesario, por ejemplo, un objeto utilizado como auxiliar. Pero

además cuando se sale de una función donde se ha creado un objeto, al abandonar el ámbito, el objeto local se destruye.

Si no se declara un destructor en una clase, el compilador crea uno por defecto pero este destructor, al igual que ocurre con el constructor por defecto, no tendrá sentido si disponemos de atributos de tipo puntero declarados dentro de la clase, puesto que la memoria reservada para estos punteros no será liberada por el destructor por defecto. Así que es recomendable declarar e implementar un destructor en toda clase que tenga algún atributo que utilice la memoria de forma dinámica, por ejemplo, un puntero, un vector, etc. Las características de un destructor son bastante similares a las de los constructores. Las más destacables son:

- No tienen ningún tipo de retorno, la función no devuelve nada.
- No tiene parámetros ni pueden ser heredados.
- Deben de ser funciones públicas para que puedan ser llamadas desde cualquier parte donde se tenga acceso al objeto.
- Se definen con el mismo nombre de la clase, pero delante del nombre se debe de poner el símbolo `~`.
- Al contrario que los constructores, no pueden ser funciones sobrecargadas, ya que al no tener parámetros ni retornos es lógico que no se puedan sobrecargar.

Siguiendo con el ejemplo de la editorial, vamos a ver cómo definimos el destructor y cómo eliminamos los atributos tipo puntero que se declaran en la clase.

```
1 class Editorial{
2
3 private:
4     char * nombre;
5     char * direccion;
6     int identificador;
7     int nPublicaciones;
8
9 public:
10    Editorial(char * name, char * address, int id, int npubli);
11    Editorial(char * name, char * address, int id);
12    Editorial(const Editorial &e);
```

```
13     ~Editorial();
14     void informacion();
15     void addPublicacion(char * titulo);
16 };
17
18 Editorial::Editorial (char * name, char * address, int id, int npubli){
19     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
20     strcpy(nombre,name);
21     direccion = new char[strlen(address)+1];
22     strcpy(direccion ,address);
23     identificador = id;
24     nPublicaciones = npubli;
25 }
26
27 Editorial::Editorial(char * name, char * address ,int id){
28     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
29     strcpy(nombre,name);
30     direccion = new char[strlen(address)+1];
31     strcpy(direccion ,address);
32     identificador = id;
33     nPublicaciones = 0;
34 }
35
36 Editorial::Editorial(const Editorial &e){
37     nombre = (char *) malloc((strlen(e.nombre)+1) * sizeof(char));
38     direccion = new char[strlen(e.direccion)+1];
39     strcpy(nombre,e.nombre);
40     strcpy(direccion ,e.direccion);
41     identificador = e.identificador;
42     nPublicaciones = e.nPublicaciones;
43 }
44
45 Editorial::~~ Editorial(){
46     delete [] direccion;
47     free(nombre);
48 }
```

Como podemos apreciar al implementar el destructor se ha liberado la memoria de dos formas distintas, una de ellas utilizando la función `free` y la otra utilizando el operador `delete`. Esto es debido a que para el atributo `direccion` hemos reservado memoria mediante el operador `new` y para el atributo `nombre` hemos utilizado la función `malloc`.

Tras ver los elementos principales de las clases en el lenguaje C++, vamos a ver un pequeño ejemplo concreto de una clase y de cómo se puede modular por ficheros la definición de la clase y de sus métodos.

2.5 Ejemplo

En este apartado vamos a ver un ejemplo completo de la definición de una clase modulada en ficheros.

Fichero `editorial.h`: En este fichero hemos definido la clase y declaramos sus funciones y atributos.

```
1 class Editorial{
2
3 private:
4     char * nombre;
5     char * direccion;
6     int identificador;
7     int nPublicaciones;
8
9 public:
10    Editorial(char * name, char * address, int id, int npubli);
11    Editorial(char * name, char * address, int id);
12    Editorial(const Editorial &e);
13    ~Editorial();
14    void informacion();
15    void addPublicacion(char * titulo);
16 };
```

Fichero `editorial.cpp`: En este fichero se implementan las funciones de la clase. Para ello hacemos uso del operador de ámbito `::`.

```
1 #include<iostream>
2 #include "editorial.h"
3
4 using namespace std;
5
6 Editorial::Editorial (char * name, char * address, int id, int npubli){
7     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
8     strcpy(nombre,name);
9     direccion = new char[strlen(address)+1];
```



```
10     strcpy(direccion , address);
11     identificador = id;
12     nPublicaciones = npubli;
13 }
14
15 Editorial::Editorial(char * name, char * address ,int id){
16     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
17     strcpy(nombre , name);
18     direccion = new char [strlen(address)+1];
19     strcpy(direccion , address);
20     identificador = id;
21     nPublicaciones = 0;
22 }
23
24 Editorial::Editorial(const Editorial &e){
25     nombre = (char *) malloc((strlen(e.nombre)+1) * sizeof(char));
26     direccion = new char [strlen(e.direccion)+1];
27     strcpy(nombre , e.nombre);
28     strcpy(direccion , e.direccion);
29     identificador = e.identificador;
30     nPublicaciones = e.nPublicaciones;
31 }
32
33 Editorial::~~Editorial(){
34     delete [] direccion;
35     free(nombre);
36 }
37
38 void Editorial::informacion(){
39     cout << "El nombre de la editorial es " << nombre << endl;
40 }
41
42 void Editorial::addPublicacion(char * titulo){
43     cout << "Se añade la publicacion con titulo " << titulo << endl;
44     nPublicaciones++;
45 }
```

Fichero main.cpp: En este fichero declaramos la función main donde creamos objetos de la clase Editorial y llamamos a las funciones que hemos definido.

```
1 #include<iostream>
2 #include "editorial.h"
3
```

```
4 int main() {  
5  
6     Editorial miEditorial("Thomson", "Calle Mayor", 1, 25);  
7     Editorial miEditorialaux("Norman", "Calle Australia", 2);  
8     Editorial miEditorialcopia(miEditorial);  
9  
10    miEditorialcopia.addPublicacion("Programacion en C++");  
11    miEditorialcopia.informacion();  
12    miEditorialaux.informacion();  
13  
14    system("pause");  
15    return 0;  
16 }
```

Capítulo 3

Herencia en C++

3.1 Introducción

La herencia en un lenguaje de programación orientado a objetos consiste en crear una nueva clase, llamada clase derivada, a partir de otra clase. La clase derivada “hereda”, es decir, obtiene todas las propiedades de la clase original y además puede añadir propiedades nuevas. La herencia es una herramienta bastante potente en muchos aspectos en el desarrollo de aplicaciones y podríamos afirmar que es el principio de la programación orientada a objetos. Entre las ventajas de la herencia nos encontramos con que una mejor organización del diseño en las aplicaciones, permite la reutilización de código y en ciertas situaciones mejora el mantenimiento de la aplicación.

Tras la definición debemos de aclarar algunos conceptos, por ejemplo, la clase derivada es la nueva clase que se crea y que hereda de una clase original, a la que llamaremos clase base. Una clase derivada puede convertirse en una clase base ya que a partir de una clase derivada se puede seguir obteniendo clases nuevas. Además, una clase derivada puede ser derivada de más de una clase base. Esto es lo que se denomina herencia múltiple o derivación múltiple. Por lo tanto, la herencia nos permite encapsular diferentes funcionalidades de un objeto bien real o imaginario, y poder vincular tales funcionalidades a otros objetos más complejos, que heredarán las características del objeto más básico y además añadirán nuevas funcionalidades propias del objeto más complejo.

Antes de introducirnos en los conceptos del lenguaje C++, vamos a mostrar un diagrama de clases de cómo se modela la herencia en tales diagramas. Supongamos un

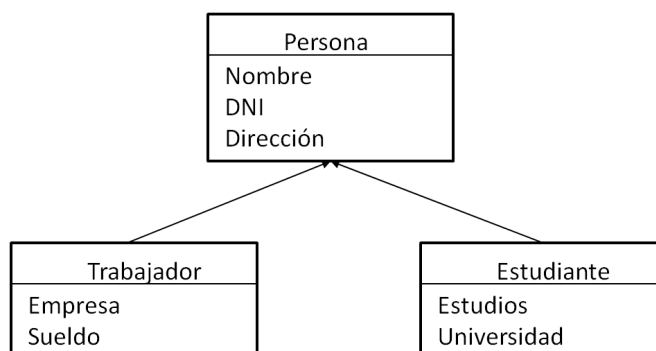


Figura 3.1: Diagrama de clases

ejemplo de una aplicación en la que debemos de registrar personas y las personas serán trabajadores o estudiantes.

El diagrama de clases de este ejemplo sería el mostrado en la Figura 3.1.

Así la clase `Persona` recoge los atributos comunes a todas las personas, pero la clase `Estudiante` además de heredar los atributos de la clase `Persona` define atributos suyos propios, igual que ocurre con la clase `Trabajador`.

3.2 Herencia simple y visibilidad en la herencia

El lenguaje C++ permite herencia simple y herencia múltiple, es decir, permite que una clase herede de una única clase o que una clase herede de más de una clase base. Sin tener en cuenta el tipo de herencia, cuando se crea una nueva clase derivada a partir de una o varias clases bases, C++ permite definir el tipo de visibilidad a aplicar en la herencia. La visibilidad en la herencia significa en qué nivel de visibilidad se “heredan” los atributos y funciones que son públicas o protegidas en la clase base. Antes de entrar en los niveles de visibilidad debemos de aclarar que toda función o atributo que es privado en la clase base, no es visible en la clase derivada. Por lo tanto, si queremos crear una jerarquía de clases y queremos que las clases derivadas hereden ciertos atributos de las clases bases, lo recomendable sería poner esos atributos y/o funciones como protegidas (`protected`).

Una vez aclarado el concepto de los atributos y funciones privados en la clase base, pasamos a ver cómo se crea una clase derivada en el lenguaje C++. La estructura para crear una clase heredada de otra es la siguiente:

```
1 class <claseDerivada>: [public|private] <claseBase> {...};
```

donde la `claseDerivada` es el nombre de la clase que va a heredar de la `claseBase`. Un aspecto a destacar es si la clase derivada hereda de la clase base de forma pública (`public`) o de forma privada (`private`). Siguiendo el ejemplo, si quisiéramos crear la clase `Trabajador` como clase derivada de `Persona`, tendríamos que declararla de la siguiente forma:

```
1 class Trabajador: public Persona {  
2     ...  
3     ...  
4 };
```

Pero, ¿qué implica heredar de una clase base de forma pública o de forma privada? La clave está en la privacidad que queramos que la clase derivada tenga con respecto a las posibles clases derivadas que pueden surgir de ella. Cuando se realiza una herencia pública, como la que hemos declarado en la clase derivada `Trabajador`, lo que estamos indicando es que todas las funciones y atributos que son públicos en la clase `Persona` pasan a ser también públicos en la clase `Trabajador`. Por lo tanto, desde donde se cree un objeto de tipo `Trabajador` se podrán utilizar las funciones públicas de la clase `Persona`. Por el contrario, si una clase derivada se crea con visibilidad privada de la clase base, esto significa que las funciones y atributos públicos de la clase base pasan a ser privados en la clase derivada, por lo cual estas funciones y atributos no podrán ser utilizados desde fuera de la clase y tampoco serán visibles a una nueva clase derivada que tome como base la clase derivada que hereda de forma privada. En otras palabras, si la clase `Trabajador` hereda de forma privada de la clase `Persona` y creamos una nueva clase derivada llamada `TrabajadorIndefinido` tomando como clase base la clase `Trabajador`, la nueva clase derivada `TrabajadorIndefinido` no tiene acceso a los atributos y funciones públicas de la clase `Persona`. Por lo tanto, antes de decidir si la herencia se realiza con visibilidad pública o privada, habrá que tener en cuenta las cuestiones que se acaban de detallar.

Tras explicar los conceptos de visibilidad, vamos a presentar el ejemplo completo de las clase `Persona`, `Trabajador` y `Estudiante` utilizando la visibilidad pública.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Persona{
6
7 protected:
8     char * direccion;
9     char * nombre;
10    char * dni;
11
12 public:
13     Persona(char * nombre, char * direccion, char * dni);
14     char * getNombre();
15     void getInfo();
16 };
17
18 class Trabajador: public Persona{
19
20 protected:
21     int sueldo;
22     char * empresa;
23
24 public:
25     Trabajador(char * nombre, char * direccion, char * dni, int sueldo, char
        * empresa);
26     int getSueldo();
27     char * getEmpresa();
28 };
29
30 class Estudiante: public Persona{
31
32 protected:
33     char * universidad;
34     char * estudios;
35
36 public:
37     Estudiante(char * nombre, char * direccion, char * dni, char * estudios,
        char * universidad);
38     char * getEstudios();
39 };
40
41 //implementación de las clases
```

```
42
43 Persona::Persona(char * _nombre, char * _direccion, char * _dni){
44     nombre = new char[strlen(_nombre)+1];
45     strcpy(nombre, _nombre);
46     direccion = new char[strlen(_direccion)+1];
47     strcpy(direccion, _direccion);
48     dni = new char[strlen(_dni)+1];
49     strcpy(dni, _dni);
50 }
51
52 char * Persona::getNombre(){return nombre;}
53
54 void Persona::getInfo(){cout << "Nombre: " << nombre << " Direccion: " <<
    direccion << endl;}
55
56 Trabajador::Trabajador(char * _nombre, char * _direccion, char * _dni, int
    _sueldo, char * _empresa): Persona(_nombre, _direccion, _dni){
57     sueldo = _sueldo;
58     empresa = new char[strlen(_empresa)+1];
59     strcpy(empresa, _empresa);
60 }
61
62 int Trabajador::getSueldo(){return sueldo;}
63
64 char * Trabajador::getEmpresa(){return empresa;}
65
66 Estudiante::Estudiante(char * _nombre, char * _direccion, char * _dni, char
    * _estudios, char * _universidad): Persona(_nombre, _direccion, _dni) {
67     estudios = new char[strlen(_estudios)+1];
68     strcpy(estudios, _estudios);
69     universidad = new char[strlen(_universidad)+1];
70     strcpy(universidad, _universidad);
71 }
72
73 char * Estudiante::getEstudios(){return estudios;}
74
75 //Vamos a crear objetos de las diferentes clases y a invocar los métodos.
76
77 int main(){
78
79     Persona maria("Maria", "Gran via", "85296385G");
80     cout << "—————" << endl;
81     maria.getInfo();
82     cout << "—————" << endl;
```

```
83
84     Estudiante pepe("Pepe","Paseo de la catedra","85697412J","Informatica","
        Universidad de Murcia");
85     cout << "—————" << endl;
86     cout << "Estudios " << pepe.getEstudios() << endl;
87     cout << "—————" << endl;
88
89     Trabajador josefa("Josefa","Avenida del rio","65473251U",2500,"Oracle
        Solutions");
90     cout << "—————" << endl;
91     josefa.getInfo();
92     cout << "Sueldo " << josefa.getSueldo() << endl;
93     cout << "—————" << endl;
94
95     system("pause");
96
97     return 0;
98 }
```

Como vemos en el ejemplo, las clases derivadas de la clase `Persona` pueden utilizar los atributos definidos como protegidos y las funciones definidas como públicas. Si los atributos hubieran sido definidos como privados no podrían ser utilizados. Por otro lado, los constructores de las clases derivadas deben de llamar al constructor de la clase base cuando sea necesario para iniciar los valores correspondientes. Esto en el ejemplo se puede apreciar en las clases `Estudiante` y `Trabajador`:

```
1 Estudiante(char * _nombre, char * _direccion, char * _dni, char * _estudios
    , char * universidad) : Persona(_nombre, _direccion, _dni) {...}
```

Sin embargo, en el ejemplo anterior, las clases derivadas no han redefinido ningún método de la clase base. Por lo tanto, cuando un objeto de la clase derivada invoca a un método de la clase base, la funcionalidad que aplica es la de la clase base, pero ¿qué ocurre si queremos redefinir un método de la clase base en una clase derivada? Para poder redefinir métodos en las clases derivadas, la clase base debe indicar qué métodos permite redefinir en las clases derivadas declarándolos como métodos virtuales. Así, un método en la clase derivada que tenga la misma signatura que un método virtual de la clase base significa que ese método está redefinido. Es importante tener en cuenta que en el fichero de cabecera (.h) de la clase derivada se deben de incluir los métodos que

se redefinen de la clase base. Hay que tener en cuenta que una vez que un método es declarado como virtual, permanece como virtual en las clases derivadas. Para invocar las diferentes versiones de los métodos se debe de utilizar la clasificación de rutinas, es decir, especificar a qué clase se está haciendo referencia de la siguiente forma:

```
1 nombreClase :: nombreMetodo
```

Así, si quisiéramos redefinir el método `getInfo()` de la clase `Persona` en sus clases derivadas, deberíamos de declarar el método `getInfo()` como `virtual` y luego redefinirlo. También hay que comentar que los métodos virtuales nos permiten utilizar el polimorfismo en el lenguaje C++, puesto que mediante estos métodos podremos aplicar la ligadura dinámica. Vamos a ver un ejemplo de cómo funciona el polimorfismo y las funciones virtuales.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Persona{
6
7 protected:
8     char * direccion;
9     char * nombre;
10    char * dni;
11
12 public:
13     Persona(char * nombre, char * direccion, char * dni);
14     char * getNombre();
15     virtual void getInfo();
16 };
17
18 class Trabajador: public Persona{
19
20 protected:
21     int sueldo;
22     char * empresa;
23
24 public:
25     Trabajador(char * nombre, char * direccion, char * dni, int sueldo, char
        * empresa);
```

```
26     int getSueldo();
27     char * getEmpresa();
28     //sobrescribiendo el método en la cabecera
29     void getInfo();
30     char * getNombre();
31 };
32
33 class Estudiante: private Persona{
34
35 protected:
36     char * universidad;
37     char * estudios;
38
39 public:
40     Estudiante(char * nombre, char * direccion, char * dni, char * estudios,
41               char * universidad);
42     char * getEstudios();
43     //sobrescribiendo el método en la cabecera
44     void getInfo();
45     char * getNombre();
46 };
47 //implementación de las clases
48
49 Persona::Persona(char * _nombre, char * _direccion, char * _dni){
50     nombre = new char[strlen(_nombre)+1];
51     strcpy(nombre, _nombre);
52     direccion = new char[strlen(_direccion)+1];
53     strcpy(direccion, _direccion);
54     dni = new char[strlen(_dni)+1];
55     strcpy(dni, _dni);
56 }
57
58 char * Persona::getNombre(){return nombre;}
59
60 void Persona::getInfo(){cout << "Nombre: " << nombre << " Direccion: " <<
61     direccion << endl;}
62
63 Trabajador::Trabajador(char * _nombre, char * _direccion, char * _dni, int
64     _sueldo, char * _empresa): Persona(_nombre, _direccion, _dni){
65     sueldo = _sueldo;
66     empresa = new char[strlen(_empresa)+1];
67     strcpy(empresa, _empresa);
68 }
```

```
67
68 int Trabajador::getSueldo(){return sueldo;}
69
70 char * Trabajador::getEmpresa(){return empresa;}
71
72 void Trabajador::getInfo(){
73     //llamando al método de la clase base
74     Persona::getInfo();
75     cout << "Sueldo " << sueldo << " Empresa " << empresa << endl;
76 }
77
78 char * Trabajador::getNombre(){
79     cout << "Soy un trabajador y mi nombre es: " << endl;
80     return Persona::nombre;
81 }
82
83 Estudiante::Estudiante(char * _nombre, char * _direccion, char * _dni, char
    * _estudios, char * _universidad): Persona(_nombre, _direccion, _dni) {
84     estudios = new char[strlen(_estudios)+1];
85     strcpy(estudios, _estudios);
86     universidad = new char[strlen(_universidad)+1];
87     strcpy(universidad, _universidad);
88 }
89
90 char * Estudiante::getEstudios(){return estudios;}
91
92 void Estudiante::getInfo(){
93     //llamando al método de la clase base
94     Persona::getInfo();
95     cout << "Estudios " << estudios << " Universidad " << universidad <<endl;
96 }
97
98 char * Estudiante::getNombre(){
99     cout << "Soy un estudiante y mi nombre es: " << endl;
100     return Persona::nombre;
101 }
102
103 //Vamos a crear objetos de las diferentes clases y a invocar los métodos.
104
105 int main(){
106
107     Persona * maria = new Estudiante("Maria", "Gran via", "85296385G", "
        Educacion", "Universidad de Valencia");
108     cout << "—————" << endl;
```

```
109  maria->getInfo ();
110  cout << "Nombre " << maria->getNombre() << endl;
111  cout << "—————" << endl;
112
113  Persona * josefa = new Trabajador("Josefa", "Avenida del rio", "65473251U"
    ,2500, "Oracle Solutions");
114  josefa->getInfo ();
115  cout << "Nombre " << josefa->getNombre() << endl;
116  cout << "—————" << endl;
117
118  system("pause");
119
120  return 0;
121 }
```

Si nos fijamos en el ejemplo, tenemos que la clase `Estudiante` y `Trabajador` han redefinido el método `getInfo()` y dicho método está declarado como `virtual` en la clase base `Persona`. Sin embargo, la clase `Trabajador` y `Estudiante` han declarado un método llamado `getNombre()`. Este método también está definido en la clase base, pero no está definido como `virtual`. Vamos a analizar la salida del programa y ver cuándo se ha aplicado la ligadura dinámica y cuándo no. La salida del programa es la siguiente:

```
—————
Nombre: Maria Direccion: Gran via
Estudios: Educacion Universidad: Universidad de Valencia
Maria
—————
Nombre: Josefa Direccion: Avenida del rio
Sueldo: 2500 Empresa: Oracle Solutions
Josefa
—————
```

Al analizar la salida podemos ver que cuando se ha llamado al método `getInfo()` sí que se han ejecutado los métodos redefinidos en las clases derivadas, sin embargo, se ha ejecutado el método `getNombre()` de la clase base `Persona`, no de las clases derivadas. Esto es así porque el método `getNombre()` no está definido como `virtual` de ahí que no se haya aplicado la ligadura dinámica. Un último aspecto a comentar es el uso de la

función `dynamic_cast` que se utiliza para comparar el tipo de un objeto (igual que la función `instanceof` del lenguaje Java). Esta función sólo es aplicable a tipos puntero. Veamos un ejemplo sencillo:

```
1 int main() {
2
3   int nInvitados=3;
4   Persona * misInvitados[nInvitados];
5
6   misInvitados[0] = new Estudiante("Maria", "Gran via", "85296385G", "
   Educacion", "Universidad de Valencia");
7
8   misInvitados[1] = new Trabajador("Josefa", "Avenida del rio", "65473251U"
   ,2500, "Oracle Solutions");
9
10  misInvitados[2] = new Trabajador("Gregoria", "Avenida las Americas", "
   32589854U",1500, "IMB Solutions");
11
12  int nEstudiante = 0;
13  for(int i = 0; i < nInvitados; i++){
14    // Si es estudiante, añado un estudiante invitado.
15    if(dynamic_cast<Estudiante *>(misInvitados[i])){
16      nEstudiante++;
17    }
18  }
19
20  cout << "Numero de estudiantes = " << nEstudiante << endl;
21  cout << "Numero de trabajadores = " << nInvitados-nEstudiante << endl;
22
23  system("pause");
24
25  return 0;
26 }
```

3.3 Herencia múltiple

Una de las características que ofrece el lenguaje C++ y que no ofrecen de forma directa otros lenguajes como Java, es que permite la herencia múltiple. El concepto de herencia múltiple consiste en crear una clase derivada heredando de una o más clases base. Un problema bastante común cuando se hereda de dos clases bases es que exista cierta

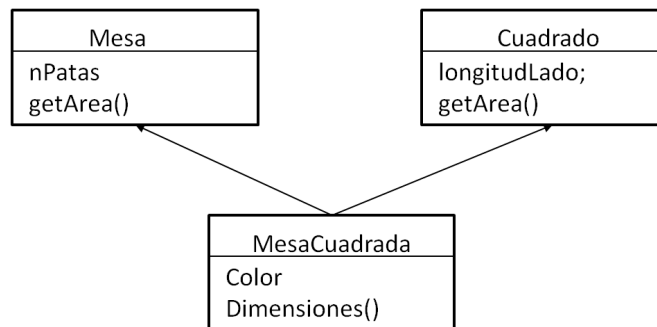


Figura 3.2: Diagrama de clases de herencia múltiple

ambigüedad, por ejemplo en dos atributos que se llamen igual o dos métodos con el mismo nombre. Para solucionar tal ambigüedad se aplica el operador de ámbito `::` indicando de qué clase base se desea ejecutar el método. Un ejemplo de herencia múltiple podría darse en el ejemplo de la Figura 3.2.

En este ejemplo tenemos una clase `MesaCuadrada` que hereda a la vez de la clase `Mesa` y de la clase `Cuadrado`. El problema comentado ocurre cuando la clase `MesaCuadrada` necesita llamar al método `getArea()`. En este caso y como hemos comentado se debe de especificar la clase de la que se quiere ejecutar el método. El ejemplo presentado en el diagrama quedaría implementado de la siguiente forma:

```

1 #include<iostream>
2
3 using namespace std;
4
5 class Mesa{
6
7 protected:
8     int nPatas;
9     int area;
10
11 public:
12     Mesa(int _nPatas, int _area){nPatas = _nPatas; area = _area;};
13     int getArea(){return area;};
14 };
15
16 class Cuadrado{
17

```

```
18 protected:
19     int longitudLado;
20
21 public:
22     Cuadrado(int _longitud){longitudLado = _longitud;};
23     int getArea(){return longitudLado*longitudLado;};
24 };
25
26 class MesaCuadrada: public Mesa, public Cuadrado{
27
28 protected:
29     char color [20];
30
31 public:
32     MesaCuadrada(char _color [20], int _longitud, int _nPatas);
33 };
34
35 //implementación del constructor de MesaCuadrada
36
37 MesaCuadrada::MesaCuadrada(char _color [20], int _longitud, int _nPatas):
38     Mesa(_nPatas, _longitud*_longitud), Cuadrado(_longitud){
39     strcpy(color, _color);
40 }
41
42 int main(){
43     MesaCuadrada mimesa("Rojo",5,4);
44
45     cout << "El area del tablero de la mesa es " << mimesa.Cuadrado::getArea
46         () << endl;
47
48     system("pause");
49
50     return 0;
51 }
```

Como hemos podido ver en el ejemplo, para definir herencia múltiple lo único que hay que indicar al declarar la nueva clase derivada son las clases de las que hereda.

```
1 class MesaCuadrada: public Mesa, public Cuadrado{...}
```

Además en el constructor de `MesaCuadrada` vemos cómo se llama a los constructores

de las clases de las que se hereda:

```
1 MesaCuadrada(char _color[20], int _longitud, int _nPatas): Mesa(_nPatas,
    _longitud*_longitud), Cuadrado(_longitud){strcpy(color, _color);}
```

Por último, para evitar la ambigüedad del método `getArea()`, se utiliza el operador de ámbito para aclarar que queremos utilizar el método de la clase `Cuadrado`. Otra posible solución podría haber sido redefinir en la clase `MesaCuadrada` el método `getArea()` especificando el método que se desea utilizar. El método `getArea()` redefinido quedaría de la siguiente forma:

```
1 // Redefinición del método getArea()
2
3 int MesaCuadrada::getArea() {
4     return Cuadrado::getArea();
5 }
6
7 int main() {
8
9     MesaCuadrada mimesa("Rojo", 5, 4);
10
11     cout << "El area del tablero de la mesa es " << mimesa.getArea() << endl;
12
13     system("pause");
14
15     return 0;
16 }
```

Al redefinir el método `getArea()` en la clase `MesaCuadrada`, cuando llamamos al método siempre se llama al de la clase `Cuadrado`.

Un último aspecto a comentar es la ambigüedad y duplicidad que se puede producir en la herencia múltiple si dos clases heredan de una clase común y después otra clase hereda de esas dos clases. Un ejemplo sería el diagrama de la Figura 3.3.

Como vemos en el ejemplo, la clase `Becario` hereda de la clase `Trabajador` y `Estudiante`, pero a su vez estas clases heredan de la clase `Persona` por lo que se podría interpretar que `Becario` hereda dos veces de la clase `Persona`. Para evitar estas ambigüedades, existe la herencia mediante clases virtuales. La herencia mediante clases virtuales tiene como función eliminar esa posible duplicidad de heredar dos veces de una

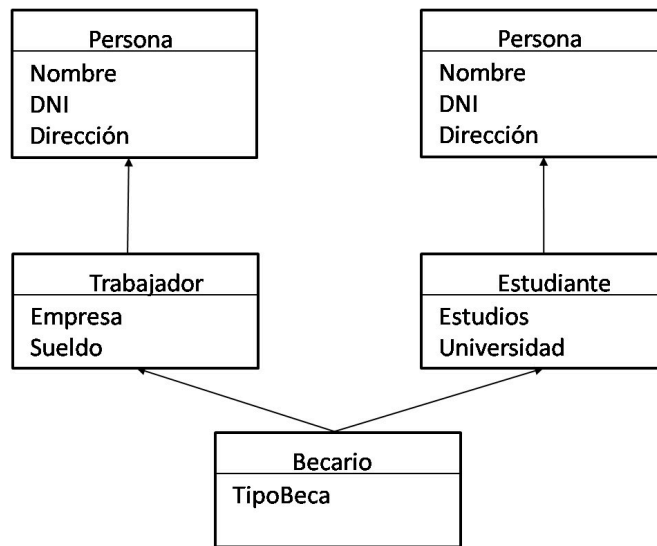


Figura 3.3: Diagrama de clases

misma clase. La única modificación en el código es al declarar la herencia en las clases `Trabajador` y `Estudiante` que heredan de la clase virtual `Persona`. La declaración de la herencia en este caso sería como sigue:

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Persona{
6
7 protected:
8     char * direccion;
9     char * nombre;
10    char * dni;
11
12 public:
13    Persona(char * nombre, char * direccion, char * dni);
14    char * getNombre();
15    void getInfo();
16 };
17
18 class Trabajador: public virtual Persona{
19
```

```
20 protected:
21     int sueldo;
22     char * empresa;
23
24 public:
25     Trabajador(char * nombre, char * direccion, char * dni, int sueldo, char
        * empresa);
26     int getSueldo();
27     char * getEmpresa();
28 };
29
30 class Estudiante : public virtual Persona{
31
32 protected:
33     char * universidad;
34     char * estudios;
35
36 public:
37     Estudiante(char * nombre, char * direccion, char * dni, char * estudios,
        char * universidad);
38     char * getEstudios();
39 };
40
41 //implementación de las clases
42
43 Persona::Persona(char * _nombre, char * _direccion, char * _dni){
44     nombre = new char[ strlen(_nombre)+1];
45     strcpy(nombre, _nombre);
46     direccion = new char[ strlen(_direccion)+1];
47     strcpy(direccion, _direccion);
48     dni = new char[ strlen(_dni)+1];
49     strcpy(dni, _dni);
50 }
51
52 char * Persona::getNombre(){return nombre;}
53
54 void Persona::getInfo(){cout << "Nombre: " << nombre << " Direccion: " <<
    direccion << endl;}
55
56 Trabajador::Trabajador(char * _nombre, char * _direccion, char * _dni, int
    _sueldo, char * _empresa): Persona(_nombre, _direccion, _dni){
57     sueldo = _sueldo;
58     empresa = new char[ strlen(_empresa)+1];
59     strcpy(empresa, _empresa);
```

```
60 }
61
62 int Trabajador::getSueldo(){return sueldo;}
63
64 char * Trabajador::getEmpresa(){return empresa;}
65
66 Estudiante::Estudiante(char * _nombre, char * _direccion, char * _dni, char
    * _estudios, char * _universidad): Persona(_nombre, _direccion, _dni) {
67     estudios = new char[strlen(_estudios)+1];
68     strcpy(estudios, _estudios);
69     universidad = new char[strlen(_universidad)+1];
70     strcpy(universidad, _universidad);
71 }
72
73 char * Estudiante::getEstudios(){return estudios;}
74
75 class Becario: public Estudiante, public Trabajador{
76
77 protected:
78     char tipoBeca[50];
79
80 public:
81     Becario(char * _nombre, char * _direccion, char * _dni, char * _estudios
        , char * _universidad, int _sueldo, char * _empresa, char _tipoBeca
        [50]);
82     char getEstudios();
83 };
84
85 //Implementación de Becario
86
87 Becario::Becario(char * _nombre, char * _direccion, char * _dni, char *
    _estudios, char * _universidad, int _sueldo, char * _empresa, char
    _tipoBeca[50]) :
88     Estudiante(_nombre, _direccion, _dni, _estudios, _universidad), Trabajador(
        _nombre, _direccion, _dni, _sueldo, _empresa), Persona(_nombre, _direccion,
        _dni){
89     strcpy(tipoBeca, _tipoBeca);
90 }
91
92 int main(){
93
94     Becario pepe("Maria", "Gran via", "85296385G", "Educacion", "Universidad de
        Valencia", 2500, "Oracle Solutions", "Tipo 1");
95     pepe.getInfo();
```

```
96     cout << "Nombre " << pepe.getNombre() << endl;
97     cout << "—————" << endl;
98
99     system("pause");
100
101     return 0;
102 }
```

Como podemos ver en el constructor de `Becario` hemos utilizado los constructores de `Persona`, `Trabajador` y `Estudiante` y además hemos modificado la forma de heredar de la clase `Trabajador` y de la clase `Estudiante`, puesto que para evitar ambigüedades se ha realizado la herencia colocando delante de la clase `Persona` la palabra `virtual`. Con esto lo que se consigue es que la clase `Becario` herede una sola vez de la clase `Persona`.

Capítulo 4

Entrada/Salida mediante Ficheros

4.1 Introducción

En este tema vamos a tratar la lectura y escritura de datos a través de ficheros, pero para entender cómo funcionan los operadores de lectura y escritura vamos a repasar de forma breve los operadores para lectura y escritura de datos por los dispositivos de entrada y salida estándar.

Para la entrada y salida de datos C++ utiliza los stream o flujos, que no son más que un canal por el cual fluyen los datos para llegar a su destino. Tradicionalmente se ha considerado que la entrada y salida estándar son el teclado y la pantalla respectivamente. De ahí que de forma estándar C++ tenga asociado al objeto `cin` la entrada estándar, es decir, la entrada de datos a partir de teclado. Para la salida estándar C++, utiliza el objeto `cout`. Además, la salida estándar de los mensajes de error lo tiene asociado al objeto `cerr`, que por defecto muestra la información por pantalla. Si comparamos estos tres objetos con el lenguaje C, veremos que los tres objetos para la entrada y salida de datos en C++ se corresponden con los objetos `stdin`, `stdout` y `stderr` del lenguaje C.

Para trabajar con la entrada y salida estándar, se debe incluir la librería de C++ `iostream`. Una vez incluida la librería se puede comenzar a hacer uso de los objetos mencionados anteriormente (`cin`, `cout`, `cerr`). Veamos un ejemplo de cómo utilizarlos:

```
1 // Se incluye la librería del estándar
2 #include <iostream>
3 int main() {
4
```

```
5 // El mensaje se imprime por pantalla.
6     std::cout << "Estoy utilizando la escritura por pantalla";
7
8 // Se muestra el error
9     std::cerr << "Esto es un error";
10
11 // Se espera leer datos de teclado.
12     std::cin.get();
13
14     return 0;
15 }
```

Un aspecto importante a recordar antes de introducirnos en la entrada y salida de datos a y desde ficheros, son los operadores de direccionamiento. Estos operadores son << y >> y se encargan de direccionar como su nombre indica el flujo de datos hacia un stream concreto. Para las salidas de datos a un stream de salida, como puede ser a fichero o a pantalla, se utiliza el operador <<. Para las entradas de datos recogidas en un flujo de entrada, por ejemplo teclado o un fichero, se utiliza el operador >>. Para ver cómo funcionan vamos a ver el siguiente ejemplo:

```
1 // Ejemplo de operadores de direccion
2 #include<iostream>
3 int main(){
4
5     char fichero [50];
6
7     //Se pide por pantalla el fichero
8     std::cout << "Que fichero quiere leer? ";
9
10    //Se lee lo que escribe el usuario por teclado
11    std::cin >> fichero;
12
13    //Se muestra por pantalla el mensaje y el contenido de la variable
14    // fichero
15    std::cout << "El fichero que quiere leer es " << fichero << " ";
16
17    system("pause");
18
19    return 0;
20 }
```

Además, es posible en una misma línea de código leer o escribir varios elementos a la vez, simplemente nombrando una vez al stream. También podemos enviar al stream un `endl` para provocar un salto de línea. Un ejemplo sería:

```
1 // Ejemplo de operadores de dirección
2 #include<iostream>
3 using namespace std;
4 int main(){
5     char nombre[50];
6     char apellidos[50];
7     char direccion[50];
8     char instruccion[50] = "separados por espacios ";
9
10    //Se muestra el mensaje por pantalla
11    cout << "Escribe tu nombre, apellidos y direccion " << instruccion;
12
13    //Se lee lo que escribe el usuario por teclado
14    cin >> nombre >> apellidos >> direccion ;
15
16    return 0;
17 }
```

4.2 Manejo de ficheros

Una vez que hemos repasado los streams de entrada y salida estándar y los operadores de direccionamiento, vamos a explicar cómo podemos leer y escribir datos de un fichero. Una característica de los streams estándar es que son creados y abiertos de forma automática, sin embargo, para trabajar con ficheros, los flujos o streams deben ser creados y abiertos antes de comenzar a trabajar con ellos.

En el lenguaje C++ para manipular ficheros debemos hacer uso de las clases `fstream`, `ifstream` y `ofstream`. Estas tres clases nos proporcionan la lectura y escritura, sólo la lectura y sólo la escritura de un fichero respectivamente.

La sintaxis para utilizar estas clases es la siguiente:

- Para lectura y escritura: Clase `fstream`
 - `fstream()`;
 - `fstream(const char * name, int mode, int = filebuf::openprot)`;

- `fstream(int);`
- `fstream(int _f, char*, int);`
- Sólo para lectura : Clase `ifstream`
 - `ifstream();`
 - `ifstream(const char * name, int mode, int = filebuf::openprot);`
 - `ifstream(int);`
 - `ifstream(int _f, char*, int);`
- Sólo para escritura : Clase `ofstream`
 - `ofstream();`
 - `ofstream(const char * name, int mode, int = filebuf::openprot);`
 - `ofstream(int);`
 - `ofstream(int _f, char*, int);`

Como podemos apreciar, cada clase tiene cuatro constructores para crear el objeto. Dependiendo del utilizado debemos de especificar unos parámetros u otros. El primer constructor crea un objeto que aún no tiene asociado un fichero, por ello para utilizar este constructor tendremos que utilizar el método `open("nombreFichero")`, para establecer la conexión entre el nombre del fichero y el stream. El segundo constructor lo que hace es crear un objeto que se asocia al fichero en disco. Al utilizar este constructor el fichero ya queda abierto y asociado al stream correspondiente, donde el primer parámetro `char *` queda apuntando a la cadena de caracteres con el nombre del fichero. El tercer constructor crea un stream pero como parámetro hay que pasarle el identificador de un fichero. Por último, el cuarto constructor crea un stream utilizando el identificador del fichero y pasándole un buffer y el tamaño de éste.

Un aspecto a comentar son los diferentes parámetros de apertura de un fichero:

- `ios::app` Para añadir información.
- `ios::ate` Para colocar el puntero a final de fichero.

- `ios::in` Para realizar una lectura del fichero. Esta opción está por defecto al usar un objeto de la clase `ifstream`.
- `ios::out` Para realizar la escritura en un fichero. Esta es la opción por defecto al usar un objeto de la clase `ofstream`.
- `ios::nocreate` Esta operación no crea el fichero si éste no existe.
- `ios::noreplace` Esta operación crea un fichero si existe uno con el mismo nombre.
- `ios::trunc` Esta operación crea un fichero y si existe uno con el mismo nombre lo borra.
- `ios::binary` Para lectura y escritura de ficheros binarios.

Como hemos comentado, los streams no son más que objetos de cierta clase y esta clase nos ofrece una serie de métodos para comprobar y consultar las condiciones y el estado en el que se encuentran. En la siguiente tabla podemos consultar algunos de los métodos más utilizados:

Método	Descripción
<code>bad</code>	Devuelve true si ha ocurrido un error
<code>clear</code>	Sirve para limpiar los flag de estado
<code>close</code>	Cierra el stream
<code>eof</code>	Indica si se ha alcanzado el final de un fichero devolviendo true
<code>fail</code>	Devuelve true si ha ocurrido un error
<code>fill</code>	Establecer manipulador de carácter de relleno
<code>flush</code>	Vaciar el buffer de un stream
<code>gcount</code>	Indica el número de caracteres leídos en la última operación de entrada
<code>get</code>	Método para ir leyendo carácter a carácter
<code>getline</code>	Método para leer una línea completa, hasta un <code>\n</code> ó <code>\r</code>
<code>ignore</code>	Método para leer y descartar caracteres
<code>open</code>	Método para abrir un stream tanto de entrada como de salida
<code>precision</code>	Manipula la precisión del stream
<code>put</code>	Método para escribir caracteres
<code>read</code>	Método para leer datos de un stream hacia un buffer
<code>seekg</code>	Método para acceder de forma aleatoria sobre un stream de entrada
<code>seekp</code>	Método para acceder de forma aleatoria sobre un stream de salida
<code>tellg</code>	Método que devuelve el puntero del stream de entrada
<code>tellp</code>	Método que devuelve el puntero del stream de salida
<code>write</code>	Método para escribir datos desde un buffer hacia un stream

Para visualizar cómo funcionan los métodos que hemos comentado y concretamente cómo se lee y se escribe en un fichero, vamos a mostrar varios ejemplos, donde encima de cada línea se indica mediante un comentario cuál es la función que realiza el método utilizado.

4.2.1 Ejemplo1: Escribiendo en un fichero

```
1 #include<iostream>
2 #include<fstream>
3 using namespace std;
4 int main(){
5     // se crea el objeto de la clase ofstream
6     ofstream mifichero;
7
8     // se abre el fichero
9     mifichero.open("ejemplo.txt");
10
11    // se escribe en el fichero
12    mifichero << "Escribo la primera línea" << endl;
13    mifichero << "Escribo la segunda línea" << endl;
14
15    // se cierra el fichero
16    mifichero.close();
17
18    return 0;
19 }
```

4.2.2 Ejemplo 1.1: Escribiendo en un fichero utilizando otro constructor

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main()
5 {
6     // se crea el objeto de la clase ofstream y se abre el fichero ,
7     // con la opción de añadir en el fichero ya existente
8     ofstream mifichero("ejemplo.txt", ios::app);
9
10    // se comprueba si hay algún error
11    if (mifichero.bad()){
```

```
12     cout << "Error al abrir el archivo";
13     return -1;
14 }
15 // se escribe en el fichero
16 mifichero << "Escribo la primera línea" << endl;
17 mifichero << "Escribo la segunda línea" << endl;
18
19 // se cierra el fichero
20 mifichero.close();
21
22 return 0;
23 }
```

4.2.3 Ejemplo 2: Leyendo de un fichero línea a línea

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(){
5     // se crea un objeto de la clase ifstream y abro el fichero
6     ifstream mifichero("ejemplo.txt");
7     char aux[128];
8
9     // se comprueba que todo es correcto
10    if(mifichero.fail()){
11        cerr << "Se ha producido un error al abrir el fichero" << endl;
12        return -1;
13    }
14
15    // se lee el fichero hasta final de fichero
16    while(!mifichero.eof()){
17        // se lee línea a línea
18        mifichero.getline(aux, sizeof(aux));
19        cout << aux << endl;
20    }
21    // se cierra el fichero
22    mifichero.close();
23
24    system("pause");
25
26    return 0;
27 }
```

4.2.4 Ejemplo 3: Leyendo un fichero y copiándolo en otro, utilizando las funciones `write` y `getline`

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main(){
6     // se crea un objeto de la clase ifstream y abre el fichero
7     ifstream mifichero("ejemploAleer.txt");
8     // se crea un objeto de la clase ofstream y abre el fichero
9     ofstream mifichero2("ejemploAcopiar.txt");
10
11     char aux[128] = "";
12
13     // se comprueba que todo es correcto
14     if(mifichero.fail()){
15         cerr << "Se ha producido un error al abrir el fichero" << endl;
16     } else{
17         // se lee el fichero hasta final de fichero
18         while(!mifichero.eof()){
19             // se lee línea a línea
20             mifichero.getline(aux, sizeof(aux), '\n');
21
22             // Conforme se lee se escribe
23             mifichero2.write(aux, sizeof(aux));
24         }
25         // se cierran los ficheros
26         mifichero.close();
27         mifichero2.close();
28     }
29
30     return 0;
31 }
```

4.2.5 Ejemplo 3.1: Leyendo un fichero y copiándolo en otro, utilizando los operadores de dirección

```
1 #include<iostream>
2 #include<fstream>
3 using namespace std;
4
```

```
5 int main(){
6     // se crea un objeto de la clase ifstream y abre el fichero
7     ifstream mifichero("ejemploAleer.txt");
8     // se crea un objeto de la clase ofstream y abre el fichero
9     ofstream mifichero2("ejemploAcopiar.txt");
10
11     char aux[128] = "";
12
13     // se comprueba que todo es correcto
14     if (mifichero.fail())
15         cerr << "Se ha producido un error al abrir el fichero" << endl;
16     else{
17         // se lee el fichero hasta final de fichero
18         while(!mifichero.eof()){
19             // se lee
20             mifichero >> aux;
21             // se escribe
22             mifichero2 << aux << endl;
23         }
24         // se cierran los ficheros
25         mifichero.close();
26         mifichero2.close();
27     }
28     return 0;
29 }
```


Parte II

LIBRERÍA GALIB

Capítulo 5

Introducción a la librería GALib

5.1 Introducción

GALib [1] es una librería de objetos implementados en el lenguaje C++ que tiene como objetivo diseñar y construir algoritmos genéticos. La librería incluye un conjunto de herramientas para poder utilizar algoritmos genéticos y resolver diferentes problemas de optimización utilizando diferentes representaciones y operadores genéticos. Las principales características que nos ofrece esta librería de algoritmos genéticos son:

- Es una librería multiplataforma (Unix, Linux, MacOS, Windows).
- Utiliza una máquina virtual paralela (PVM) para poder trabajar con implementaciones paralelas y distribuidas.
- Permite la parametrización tanto por línea de comandos, como por archivo o en el propio código.
- Permite obtener diferentes estadísticas mediante archivos estructurados tanto de forma “on-line” como “off-line”.

De forma general para utilizar la librería se trabaja con dos clases principalmente, la clase `GAGenome` que define al cromosoma, individuo o genoma (términos que usaremos indistintamente a lo largo del documento), y la clase `GAGeneticAlgorithm` que define los diferentes elementos del algoritmo genético. Así, para definir un algoritmo genético utilizando la librería GALib se deben de seguir estos tres pasos:

- Definir la representación del cromosoma.
- Definir los operadores genéticos más adecuados.
- Definir la función objetivo, también conocida como función “fitness”.

Para los dos primeros puntos, la librería GALib dispone de varias implementaciones, sin embargo la función objetivo debe de ser definida por el usuario. A lo largo de los siguientes apartados se irán desarrollando y detallando los diferentes tipos de representación así como los diferentes tipos de algoritmos genéticos disponibles. Pero antes de describir la funcionalidad de esta librería, hay que comentar cómo se instala y cómo se comienza a utilizar la librería GALib.

5.2 Creación de un proyecto que use GALib

En este apartado nos vamos a centrar en detallar cómo configurar e iniciar un nuevo proyecto para poder utilizar la librería GALib. Aunque existen muchos entornos de programación para el lenguaje C++ y en todos ellos es posible configurar la librería, nos vamos a centrar en configurar la librería en el entorno de programación Dev-C++. También hay que tener en cuenta que si el programador prefiere no utilizar ningún entorno de programación, deberá compilar desde la línea de comandos o mediante un archivo Makefile. En este caso deberá de realizar los includes necesarios según las partes de la librería que desee utilizar.

5.2.1 Instalación de la librería GALib en el entorno Dev-C++

Antes de comenzar a configurar la instalación de la librería GALib en el entorno Dev-C++ debemos de proceder a descargar el código fuente de la misma desde su página oficial “<http://lancet.mit.edu/ga/>”. Una vez que tenemos descargado el código, debemos descomprimirlo en una carpeta que en este caso llamaremos “galib247”. Al descomprimir el código dentro de la carpeta “galib247” queda el fichero “galib.a” y una carpeta “galib247” (Figura 5.1). Dentro de esta última carpeta “galib247” es donde podemos encontrar el resto de fuentes de la librería, en concreto dentro de la carpeta llamada “ga”. Una vez que la librería está descargada y descomprimida, vamos a ver los pasos que hay que seguir para configurarla dentro de un proyecto en el entorno Dev-C++.

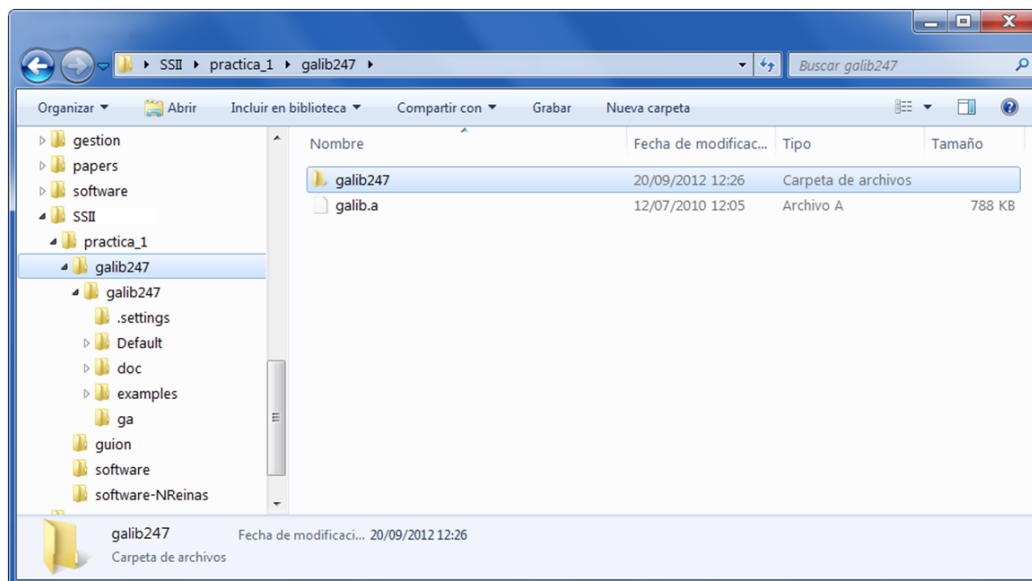


Figura 5.1: Descarga de la librería GALib

El primer paso es crear un proyecto vacío en el entorno Dev-C++. Una vez que tenemos el proyecto vacío creado, vamos a configurarlo para poder utilizar la librería GALib. Dentro de las opciones del proyecto, en la parte superior podemos encontrar diversas pestañas (archivos, principal, compilador, parámetros, directorios, etc.). Primeramente debemos de ir a la pestaña de parámetros y añadir en el cuadro de “Linker” la librería estática “galib.a” a las librerías estáticas. Para realizar esta inclusión debemos de pulsar en añadir biblioteca, ir donde se ha descomprimido la librería y seleccionar el archivo “galib.a”. Tras incluir el archivo debe de quedar la librería incluida tal como se muestra en la Figura 5.2.

Una vez incluida la librería, debemos de indicarle el path donde se encuentran los ficheros “.h” de la librería. Para indicarle el path de los includes, hay que seleccionar la pestaña de “Directorios” y situarse en la segunda pestaña “directorio de includes”. En este último paso debe quedar una configuración similar a la que aparece en la Figura 5.3.

Tras configurar todas las opciones indicadas sólo queda aceptar las modificaciones. En este punto ya podemos comenzar a crear un algoritmo genético utilizando todas las herramientas que nos ofrece esta librería.

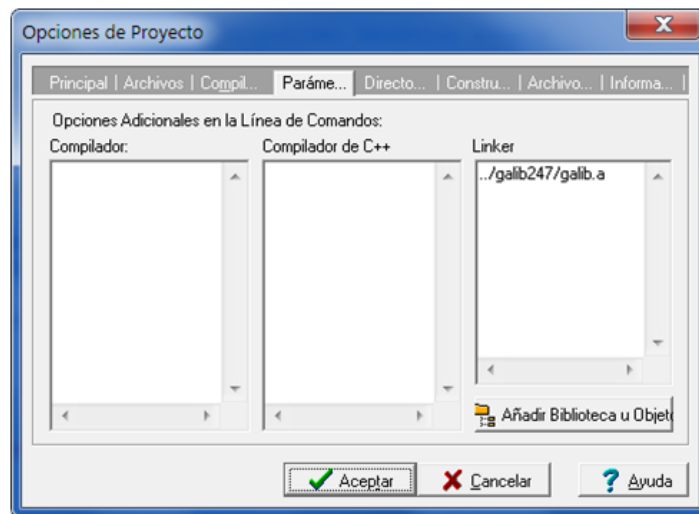


Figura 5.2: Incluyendo la librería “galib.a”

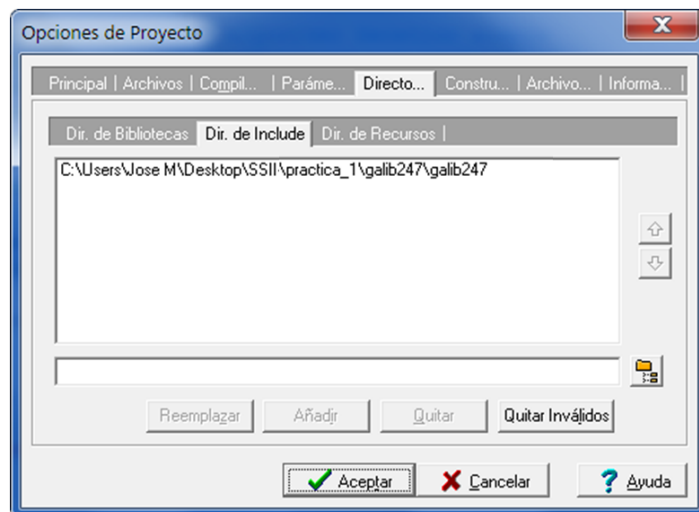


Figura 5.3: Incluyendo el path de los includes de GALib

5.2.2 Esquema básico de un algoritmo genético utilizando GALib

Cómo hemos comentado, el objetivo principal de la librería GALib es ofrecer un conjunto de herramientas para poder implementar de forma sencilla un algoritmo genético. Para resolver un problema usando un algoritmo genético implementado usando la librería GALib, debemos de elegir cómo va a ser representada la solución de nuestro problema en una estructura de datos, es decir, debemos de escoger la estructura del cromosoma que

en la librería se conoce como GAGenome. Tras decidir la representación a utilizar dependiendo del problema, hay que crear un algoritmo genético que trabaje con una población de cromosomas o individuos con la estructura previamente definida. Después mediante los diversos operadores genéticos y una función objetivo, que debe de ser definida por el usuario para evaluar cada uno de los individuos, el algoritmo quedará definido para obtener una solución al problema planteado. Por lo tanto, un proyecto debe de tener como mínimo la siguiente estructura básica:

```
1 //ESTRUCTURA BÁSICA DE UN PROYECTO UTILIZANDO GAlib
2
3 float Objective(GAGenome &);
4 int main(){
5
6     //Definición del cromosoma que en este ejemplo es de tipo string
7     binario
8
9     GA1DBinaryStringGenome genome(length, Objective);
10
11     //Definición del algoritmo genético que en este ejemplo es un algoritmo
12     genético simple
13
14     GASimpleGA ga(genome);
15
16     //Definición de los parámetros del algoritmo
17
18     // popsize = Tamaño de la población
19     ga.populationSize(popsiz);
20     // ngen = Número de generaciones
21     ga.nGenerations(ngen);
22     // pmut = Probabilidad de mutación
23     ga.pMutation(pmut);
24     // pcross = Probabilidad de cruce
25     ga.pCrossover(pcross);
26
27     //Definición del tipo de cruce a aplicar
28
29     ga.crossover(GA1DBinaryStringGenome::UniformCrossover);
30
31     //Definición de tipo de selección
32
33     GATournamentSelector sel;
34     ga.selector(sel);
```

```
33
34     //Llamada al método para que el algoritmo comience a evolucionar
35
36     ga.evolve();
37
38     //Impresión de las estadísticas una vez terminado el algoritmo genético
39
40     cout << ga.statistics() << endl;
41 }
42
43 //Definición de la función objetivo que debe de ser siempre definida por el
44     usuario
45 float Objective(GAGenome &){
46     ...
47 }
```

Como se puede apreciar, tras definir el tipo de representación del cromosoma y crear el objeto que define el algoritmo genético, solamente hay que definir los parámetros configurables de todo algoritmo genético, como el tamaño de la población, la probabilidad de cruce, de mutación y el número de generaciones. Después se definen los operadores genéticos que queremos que se apliquen y por último se llama al método `evolve()` para que se ejecute el algoritmo.

En los siguientes apartados se comentará más en detalle las posibles representaciones de los cromosomas, los diferentes tipos de operadores genéticos y los tipos de algoritmos genéticos que esta librería nos ofrece.

5.3 Descripción de los problemas utilizados

Dado que a lo largo de esta parte vamos a presentar los distintos elementos que la librería `GAlib` nos proporciona para implementar un algoritmo genético, usaremos distintos ejemplos para ilustrar dichos elementos. En esta sección definimos estos ejemplos para posteriormente poder usarlos directamente en las distintas secciones hasta completar su resolución mediante algoritmos genéticos apropiados a cada uno de ellos.

5.3.1 Optimización de una función

Como primer ejemplo, veremos un problema que consiste en minimizar la siguiente función sujeta a ciertas restricciones:

$$\begin{aligned} \text{Min } & y = x_1^2 - 2 \cdot x_1 + x_2^2 + 2 \\ \text{s.a: } & \\ & 0 \leq x_1 \leq 5 \\ & 0 \leq x_2 \leq 5 \end{aligned}$$

5.3.2 Problema de las N reinas

Otro ejemplo que resolveremos será el problema de las N reinas que consiste en situar N reinas en un tablero de ajedrez de $N \times N$ sin que se amenacen entre ellas. Una reina amenaza a otra si está en la misma fila, columna o diagonal.

Los posibles movimientos de una reina son los mostrados en la Figura 5.4.

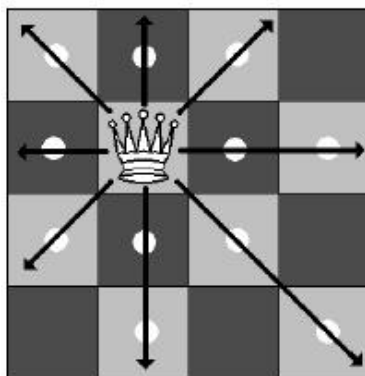


Figura 5.4: Movimientos de una reina

5.3.3 Problema de la mochila

El último ejemplo con el que trabajaremos será el problema de la mochila.

Supongamos que tenemos n tipos distintos de ítems, que van del 1 al n . De cada tipo de ítem se tienen q_i ítems disponibles, donde q_i es un entero positivo que cumple $1 \leq q_i \leq \infty$.

Cada tipo de ítem i tiene un beneficio asociado dado por v_i y un peso (o volumen) w_i . Usualmente se asume que el beneficio y el peso no son negativos.

Por otro lado se tiene una mochila, donde se pueden introducir los ítems, que soporta un peso máximo (o volumen máximo) W .

El problema consiste en meter en la mochila ítems de tal forma que se maximice el beneficio total de los ítems que contiene siempre que no se supere el peso máximo que puede soportar la misma. La solución al problema vendrá dada por la secuencia de variables x_1, x_2, \dots, x_n donde el valor de x_i indica cuántas copias se meterán en la mochila del tipo de ítem i .

El problema se puede expresar matemáticamente por medio del siguiente programa lineal:

$$\begin{aligned} \text{Max} \quad & \sum_{i=1}^n v_i x_i \\ \text{s.a:} \quad & \sum_{i=1}^n w_i x_i \leq W \\ & 1 \leq q_i \leq \infty \end{aligned}$$

Si $q_i = 1$ para $i = 1, 2, \dots, n$ se dice que se trata del problema de la mochila 0-1. Si uno o más q_i es infinito entonces se dice que se trata del problema de la mochila no acotado también llamado a veces problema de la mochila entera. En otro caso se dice que se trata del problema de la mochila acotado. En adelante, siempre trataremos con el problema de la mochila 0-1.

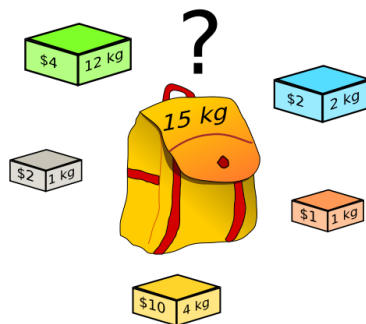


Figura 5.5: Mochila 0-1

5.4 Generadores aleatorios y semillas

La librería GALib incluye diversos métodos para generar números aleatorios. Antes de utilizar cualquier método para generar números aleatorios tenemos la posibilidad de

establecer una semilla para el generador. El método para establecer la semilla es `void GARandomSeed(unsigned s = 0)`. Si este método se llama sin parámetros o con valor 0, la semilla que se establece es la hora actual multiplicada por el ID que el sistema haya asignado al proceso. Cuando la librería se utiliza en sistemas donde no se asigna un ID a los procesos, solamente se utiliza como semilla el tiempo. Para establecer una semilla definida por el usuario, simplemente hay que pasar tal valor como parámetro al método. Si durante la ejecución del proceso se quiere cambiar de semilla, hay que llamar otra vez al método con un nuevo valor de la semilla (si se llama con el mismo valor la llamada no tiene efecto) y a partir de ese momento la semilla se verá reinicializada. Si queremos reestablecer el generador con la misma semilla, debemos realizar una llamada al método `void GAResetRNG(unsigned int s)` con un valor distinto de 0.

Los métodos disponibles en la librería para obtener números aleatorios son los siguientes:

- `int GARandomInt()`
- `int GARandomInt(int min, int max)`
- `double GARandomDouble()`
- `double GARandomDouble(int min, int max)`
- `float GARandomFloat()`
- `float GARandomFloat(int min, int max)`
- `int GARandomBit()`
- `GABoolean GAFlipCoin(float p)`
- `int GAGaussianInt(int stddev)`
- `float GAGaussianFloat(float stddev)`
- `double GAGaussianDouble(double stddev)`
- `double GAUnitGaussian()`

Los métodos `GARandomInt`, `GARandomDouble`, `GARandomFloat` que no reciben parámetros devuelven un número aleatorio en el intervalo 0 y 1. Por el contrario, los métodos `GARandomInt`, `GARandomDouble`, `GARandomFloat` que reciben los parámetros `min` y `max`, devuelven un número aleatorio entre el valor mínimo y máximo establecido, incluyendo ambos valores.

El método `GAFlipCoin` devuelve un valor booleano basado en un lanzamiento de moneda con sesgo p . Por ejemplo, si p es 1, el método siempre devuelve 1, si p es 0.75 el método devuelve 1 con una probabilidad del 75 %. Si el usuario quiere utilizar este tipo de valores aleatorios, el método más eficiente es `GARandomBit`. Este método utiliza el método numérico descrito en [5].

El resto de métodos toman como base una función de Gauss y devuelven un número aleatorio a partir de una distribución gaussiana tomando como desviación típica la que se indica como parámetro en dichos métodos. Dependiendo de la precisión necesaria en el número aleatorio, el método elegido será diferente, siendo el método `GAGaussianDouble` la más precisa. Por último, el método `GAUnitGaussian` devuelve un número aleatorio a partir de una distribución de Gauss de media 0 y desviación 1.

Sin tener en cuenta el método utilizado para obtener números aleatorios, es importante recordar que se debe siempre establecer una semilla distinta de 0 antes de comenzar con la ejecución del algoritmo para así poder reproducir una ejecución del mismo, bien para obtener de nuevo la solución o para ir ajustando los parámetros del mismo.

Capítulo 6

Clases GAGenome

6.1 Introducción

La elección de una buena representación de los individuos en un algoritmo genético es un factor clave para asegurar el éxito de la solución. La representación elegida debe representar soluciones factibles del problema. En caso de que la representación de un individuo pueda generar soluciones no factibles, la función objetivo debe ser diseñada para detectar aquellas soluciones que no sean factibles y penalizarlas de alguna manera.

La librería GALib ofrece una serie de clases para representar los individuos. Todas estas clases heredan de la clase GAGenome. Esta es la clase principal a partir de la cual se puede elegir la estructura de datos que puede tener un cromosoma y no puede ser instanciada. Las clases que definen las diferentes representaciones que la librería GALib ofrece se muestran en la Figura 6.1.

En general, podemos decir que los cuatro tipos principales de representación de los cromosomas son:

- `GAArrayGenome`: Esta clase se utiliza para representar un cromosoma mediante un array de objetos que puede representarse con o sin alelos.
- `GABinaryStringGenome`: Esta clase es utilizada para representar un cromosoma de forma binaria.
- `GAListGenome`: Esta clase es para crear un cromosoma donde es necesario tener cierto orden y la longitud de los cromosomas puede ser variable.

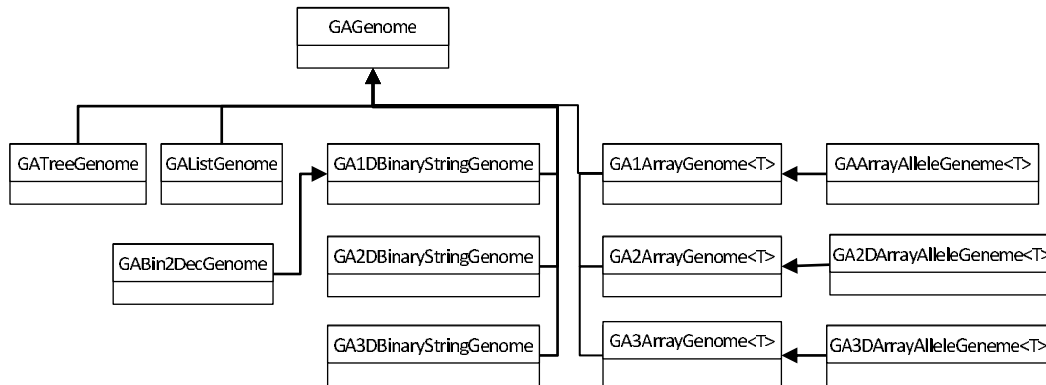


Figura 6.1: Clases para representar los cromosomas

- `GATreeGenome`: Esta clase es para representar un cromosoma utilizando la estructura de datos de un árbol.

En los siguientes apartados vamos a estudiar un poco más en detalle los métodos que tienen en común las diferentes clases que definen las representaciones de individuos disponibles en la librería `GAlib` y además vamos a estudiar en profundidad algunas de estas representaciones, en concreto, las clases `GABinaryStringGenome` y `GAArrayGenome` para dimensión 1.

6.2 Métodos principales

Sin tener en cuenta el tipo de representación elegida para resolver un problema, hay algunos métodos que son comunes a todas las representaciones. Entre ellos podemos destacar:

- `GAGenomeInitializer initializer() const / GAGenomeInitializer initializer(GAGenome::Initializer func)`: Este método devuelve/establece el operador de inicialización.
- `GAGenome::Mutator mutator() const / GAGenome::Mutator mutator(GAGenome::Mutator func)`: El método devuelve/establece el operador de mutación que se va a aplicar durante el algoritmo genético.

- `GAGenome::SexualCrossover crossover(GAGenome::SexualCrossover f) / GAGenome::AsexualCrossover crossover(GAGenome::AsexualCrossover f):` Este método establece el operador genético de cruce para un individuo.
- `void * userData() / void * userData(void * data):` Cada individuo tiene la posibilidad de contener un puntero genérico a los datos que son necesarios asociar al mismo. Este método devuelve/establece dicho puntero. Sin embargo, hay que tener en cuenta que cuando se clona un individuo, los datos asociados no se clonan por lo que la referencia es la misma.
- `int length() const / int length(int l):` Este método devuelve/establece la longitud del individuo.

Como acabamos de describir, los métodos `initializer`, `mutator` y `crossover` permiten cambiar el operador genético por defecto correspondiente. La librería `GALib` asigna operadores por defecto para cada clase `GAGenome` y usando los métodos anteriores, estos valores por defecto pueden ser modificados por otros definidos en la librería o incluso por operadores definidos por el usuario.

Para definir un operador, el usuario debe de implementar dicho operador en una función con las siguientes firmas:

- `void operadorInicio(GAGenome &g)` para el operador de inicio, donde el parámetro `g` es el genoma que se inicializa. Además, habrá que indicar que se quiere usar dicho operador usando el método `initializer`, `genoma.initializer(operadorInicio);`, donde `genoma` es un individuo del tipo que se usa en el problema que se está resolviendo.
- `int operadorMutacion(GAGenome &g, float pmut)` para el operador de mutación, donde `g` es el genoma a mutar y `pmut` es la probabilidad de mutación. Además habrá que indicar que se quiere usar dicho operador, `genoma.mutator(operadorMutacion);`.
- `int operadorCruce(const GAGenome& p1, const GAGenome& p2, GAGenome* c1, GAGenome* c2)` para el operador de cruce, donde `p1` y `p2` son los genomas a cruzar (progenitores) y `c1` y `c2`, los individuos generados. Además habrá que indicar que se quiere usar dicho operador, `genoma.crossover(operadorCruce);`.

6.3 Clase GA1DBinaryStringGenome

La clase `GA1DBinaryStringGenome` es una clase derivada de las clases `GABinaryString` y `GAGenome`. Esta clase representa al individuo como un string de ceros y unos. Los genes para este individuo son bits y los alelos para cada bit son 0 y 1.

6.3.1 Constructor de clase

Para crear un objeto de esta clase se debe usar uno de los siguientes constructores:

- `GA1DBinaryStringGenome(unsigned int x, GAGenome:: Evaluator objective = NULL, void *userData = NULL)`: Este constructor toma como argumentos la longitud del individuo, la función objetivo y se puede especificar datos asociados al individuo si se considera necesario. Si no se quieren especificar estos datos se debe de indicar el valor `NULL`.
- `GA1DBinaryStringGenome(const GA1DBinaryStringGenome&)`: Con este constructor se crea un individuo como copia del pasado como parámetro.

6.3.2 Operadores genéticos por defecto y disponibles

Los operadores genéticos disponibles en la librería para este tipo de individuos son los siguientes:

- `GA1DBinaryStringGenome::UniformInitializer`: Operador de inicio que inicializa el genoma siguiendo una distribución uniforme.
- `GA1DBinaryStringGenome::SetInitializer`: Operador de inicio que inicializa con unos.
- `GA1DBinaryStringGenome::UnsetInitializer`: Operador de inicio que inicializa con ceros.
- `GA1DBinaryStringGenome::FlipMutator`: Operador de mutación que cambia el valor de un elemento del genoma a otro de sus posibles valores.
- `GA1DBinaryStringGenome::UniformCrossover`: Operador de cruce que compara los genes de los padres y los intercambia con cierta probabilidad si son distintos.

- `GA1DBinaryStringGenome::EvenOddCrossover`: Operador de cruce en el que uno de los hijos hereda los genes pares de los padres y otro los impares.
- `GA1DBinaryStringGenome::OnePointCrossover`: Operador de cruce por un punto.
- `GA1DBinaryStringGenome::TwoPointCrossover`: Operador de cruce por dos puntos.

Por defecto, esta representación de los individuos tiene los siguientes operadores genéticos:

- El operador de inicio por defecto es `GA1DBinaryStringGenome::UniformInitializer`.
- El operador de mutación por defecto es `GA1DBinaryStringGenome::FlipMutator`.
- El operador de cruce por defecto es `GA1DBinaryStringGenome::OnePointCrossover`.

Si se quisiera cambiar el operador por defecto, usaremos los métodos `initializer`, `mutator` y `crossover` comentados en la Sección 6.2. Por ejemplo, si quisiéramos cambiar el operador de cruce a `GA1DBinaryStringGenome::TwoPointCrossover`, usaríamos el siguiente código:

```
1 genome.crossover(GA1DBinaryStringGenome::TwoPointCrossover);
```

Algunos de los métodos más comunes de esta representación de individuos son los siguientes:

- `void copy(const GA1DBinaryStringGenome & original, unsigned int dest, unsigned int src, unsigned int length)`: copia desde la posición `src`, `length` bits desde el individuo `original` al individuo al que se aplica el método.
- `short gene(unsigned int x=0) const /short gene(unsigned int, short value)`: devuelve/establece el valor de un gen.

6.3.3 Representación para el problema de minimizar una función con restricciones

Para resolver este problema donde debemos encontrar los valores X_1 y X_2 que minimizan la función $Y = X_1^2 - 2 \cdot X_1 + X_2^2 + 2$, usaremos un individuo del tipo `GA1DBinaryStringGenome` de tamaño 32 bits. Usaremos 16 bits para obtener el valor de X_1 y otros 16 bits para obtener el valor de X_2 . Cuanto mayor sea la cantidad de bits asignados a cada variable mayor será la precisión de las mismas.

Dado que el mayor valor decimal que se puede representar con 16 bits es 65535, a partir de los 16 primeros bits del individuo obtenemos el valor de X_1 (obtenemos el fenotipo desde el genotipo) como:

$$0 + \left(\frac{\text{decimalPrimeros16Bits}}{65535} \right) \times (5 - 0)$$

donde *decimalPrimeros16Bits* es el valor decimal representado en los 16 primeros bits del individuo y mediante la anterior expresión es trasladado al intervalo $[0, 5]$ al que debe pertenecer X_1 . Lo mismo haremos para obtener el valor de X_2 pero en este caso a partir de los últimos 16 bits del individuo:

$$0 + \left(\frac{\text{decimalUltimos16Bits}}{65535} \right) \times (5 - 0)$$

De esta forma la función que obtendría el fenotipo a partir del genotipo de un individuo, es decir, el valor de X_1 y X_2 , sería:

```

1 float * decodificar(GAGenome & g){
2
3     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
4
5     float * vect = new float [2];
6
7     // del gen 0 al 15 representa el valor de X1
8     float parte1 = 0;
9     for(int i=0; i<16; i++)
10         parte1+=(genome.gene(i) * pow(2.0, (float)15-i));
11     float X1 = 0 + (parte1/65535.0) * (5-0);
12
13     // del gen 16 al 31 representa el valor de X2
14     float parte2 = 0;
```



```
15 for(int i=16; i<32; i++)
16     parte2+=(genome.gene(i)* pow(2.0,(float)31-i));
17 float X2 = 0 + (parte2/65535.0) * (5-0);
18
19 vect[0] = X1;
20 vect[1] = X2;
21
22 return vect;
23 }
```

El código que crearía un individuo con esta estructura es el que se muestra a continuación:

```
1 int main(int argc , char **argv){
2
3     cout << "Este programa encuentra el valor minimo en la funcion\n";
4     cout << " y = x1^2 - 2x^1 + x2^2 + 2\n";
5     cout << "con las restricciones\n";
6     cout << "      0 <= x1 <= 5\n";
7     cout << "      0 <= x2 <= 5\n";
8     cout << "\n\n";
9     cout.flush();
10
11     ...
12
13 // El genoma contiene 16 bits para representar el valor de X1 y
14 // 16 bits para representar el valor de X2
15
16     int tam = 32;
17
18 // del gen 0 al 15 representa el valor de X1
19 // del gen 16 al 31 representa el valor de X2
20
21 // Creamos el individuo que es usado por el GA para clonarlo y crear una
22 // población de genomas.
23 GA1DBinaryStringGenome genome(tam, Objective ,NULL);
24
25     ...
26 }
```

Como se observa en el ejemplo, el individuo se crea mediante el primer constructor

comentado, indicando el tamaño que en este caso es un tamaño fijo de 32 bits de longitud, la función objetivo y, como en este caso no es necesario añadir datos al individuo, el último parámetro es `NULL`.

6.4 Clase `GA1DArrayGenome<T>`

Este tipo de representación de individuos consiste en un array genérico de una dimensión de tamaño ampliable. La clase `GA1DArrayGenome<T>` es una clase plantilla que deriva de la clase principal `GAGenome`. Cada elemento del array representa un gen y el valor de cada gen es determinado por el tipo `<T>` que se especifique. Por ejemplo, si se especifica `<int>`, cada gen representará un número entero, si por el contrario se especifica `<double>` cada gen representará un valor real.

6.4.1 Constructor de clase

Para construir un individuo utilizando este tipo de representación, la clase dispone de los siguientes constructores:

- `GA1DArrayGenome(unsigned int length, GAGenome::Evaluator objective = NULL, void * userData = NULL)`: Los argumentos del constructor son la longitud del array, la función objetivo y los datos adicionales. Si algunos de los dos últimos argumentos no se quieren indicar se deben de poner a `NULL`.
- `GA1DArrayGenome(const GA1DArrayGenome<T> &)`: Este constructor crea un individuo como copia del pasado como parámetro.

6.4.2 Operadores genéticos por defecto y disponibles

Los operadores genéticos que se encuentran disponibles para ser utilizados por este tipo de representación son los siguientes:

- `GA1DArrayGenome<>::SwapMutator`: Operador de mutación que intercambia dos genes del individuo.
- `GA1DArrayGenome<>::UniformCrossover`: Operador de cruce que compara los genes de los padres y los intercambia con cierta probabilidad si son distintos.

- `GA1DArrayGenome<>::EvenOddCrossover`: Operador de cruce en el que uno de los hijos hereda los genes pares de los padres y otro los impares.
- `GA1DArrayGenome<>::OnePointCrossover`: Operador de cruce por un punto.
- `GA1DArrayGenome<>::TwoPointCrossover`: Operador de cruce por dos puntos.
- `GA1DArrayGenome<>::PartialMatchCrossover`: Operador de cruce en el que dados dos puntos, se intercambia la parte intermedia de los dos padres.
- `GA1DArrayGenome<>::OrderCrossover`: Operador de cruce usado cuando los genomas son listas ordenadas. Se selecciona un punto de cruce, y cada hijo toma la primera parte de un padre y la segunda parte del hijo se ordena según el orden del otro padre.

Los operadores genéticos establecidos por defecto para esta representación son:

- Para inicializar el individuo se utiliza por defecto el operador `GAGenome::NoInitializer`.
- El operador de mutación establecido por defecto es `GA1DArrayGenome<>::SwapMutator`.
- El operador de cruce definido por defecto es `GA1DArrayGenome<>::OnePointCrossover`.

Por último, comentar que algunos de los métodos que pueden ser utilizados con más frecuencia cuando se hace uso de este tipo de representación de individuos son los siguientes:

- `void copy(const GA1DArrayGenome<T>& original, unsigned int dest, unsigned int src, unsigned int length)`: copia desde la posición `src`, `length` bits desde el individuo `original` al individuo al que se aplica el método.
- `T & gene(unsigned int x=0) const / T & gene(unsigned int, const T& value) const`: que devuelve/establece el valor de un gen.

6.5 Clase `GA1DArrayAlleleGenome<T>`

La clase `GA1DArrayAlleleGenome<T>` es derivada de la clase comentada anteriormente, `GA1DArrayGenome<T>`. Comparte el mismo comportamiento y añade la característica de asociar a los genes del individuo un conjunto de alelos. El valor que puede tomar cada elemento en un individuo con este tipo de representación depende del conjunto de alelos que define los posibles valores.

Si se crea un individuo con un único conjunto de alelos, el individuo tendrá la longitud que especifique el usuario y el conjunto de alelos será utilizado como posibles valores para cada gen. Si se crea el individuo utilizando un array de conjuntos de alelos, el individuo tendrá una longitud igual al número de conjuntos de alelos del array y a cada gen se le asignará un valor del conjunto de alelos correspondiente.

6.5.1 Constructor de clase

Para crear un individuo utilizando esta representación, la clase ofrece los siguientes constructores:

- `GA1DArrayAlleleGenome(unsigned int length, const GAAlleleSet <T>& alleleset, GAGenome::Evaluator objective = NULL, void * userData = NULL)`: Este constructor toma como parámetro la longitud del individuo, un conjunto de alelos, la función objetivo y los datos adicionales.
- `GA1DArrayAlleleGenome(const GAAlleleSetArray<T>& allelesets, GAGenome::Evaluator objective = NULL, void * userData = NULL)`: El constructor toma como argumentos el array de conjuntos de alelos, la función objetivo y los datos adicionales. En este constructor no se especifica la longitud del individuo.
- `GA1DArrayAlleleGenome(const GA1DArrayAlleleGenome<T>&)`: Este método crea un individuo como copia del individuo que se pasa como parámetro.

6.5.2 Definición de los alelos: Clase `GAAlleleSet<T>`

Antes de crear un individuo de este tipo, es necesario definir los conjuntos de alelos que definirán los valores de sus genes. Para ello, creamos una instancia de la clase

`GAAlleleSet<T>` y añadiremos los distintos valores al conjunto usando el método siguiente:

- `T add(const T& allele)`: El método añade el alelo `allele` al conjunto.

6.5.3 Operadores genéticos por defecto y disponibles

Al utilizar esta representación, los operadores genéticos que hay disponibles, además de los heredados de la clase `GA1DArrayGenome<T>` son:

- `GA1DArrayAlleleGenome<>::UniformInitializer`: Operador de inicio que inicializa el genoma siguiendo una distribución uniforme.
- `GA1DArrayAlleleGenome<>::OrderedInitializer`: Operador de inicio que tiene en cuenta que el genoma es una lista ordenada.
- `GA1DArrayAlleleGenome<>::FlipMutator`: Operador de mutación que cambia el valor de un elemento del genoma a otro de sus posibles valores.

Los operadores por defecto son:

- Como operador de inicialización el operador `GA1DArrayAlleleGenome<>::UniformInitializer`.
- El operador de mutación establecido por defecto es `GA1DArrayAlleleGenome<>::FlipMutator`.
- Como operador de cruce, el establecido por defecto es `GA1DArrayGenome<>::OnePointCrossover`.

6.5.4 Representación para el problema de las N reinas

Usaremos este tipo de representación para resolver el problema de las N reinas. Para ello supondremos que cada gen del individuo representa una columna del tablero de ajedrez y su valor indica la fila en la que se encuentra situada la reina de esa columna (cualquier solución al problema no tendrá más de una reina por columna). El valor por lo tanto de cada gen estará limitado al número de filas del tablero y por lo tanto podemos construir un conjunto de alelos formado por los valores enteros comprendidos

en el intervalo $[0, \text{numeroFilas}-1]$. Por lo tanto, el individuo será un array de 1 dimensión de valores enteros con una longitud igual al número de columnas del tablero y donde cada gen toma valores del conjunto de alelos comentado.

```
1 int main(int argc , char **argv)
2 {
3
4     int nreinas = 8;
5
6     cout << "Problema de las " << nreinas << " reinas \n\n";
7     cout.flush();
8
9     ...
10
11     // Conjunto enumerado de alelos —> valores posibles de cada gen del
12     // genoma
13
14     GAAlleleSet<int> alelos;
15     for(int i=0;i<nreinas;i++) alelos.add(i);
16
17     // Creamos el individuo pasándole como argumentos la longitud, el
18     // conjunto de alelos y la referencia a la función objetivo
19
20     GA1DArrayAlleleGenome<int> genome(nreinas , alelos , Objective , NULL);
21     ...
22 }
```

6.6 Función Objetivo/Fitness

Como ya hemos comentado, la librería GAlib nos ofrece un conjunto de métodos y clases para poder resolver un problema mediante un algoritmo genético. Utilizando dichos métodos y clases se puede implementar un algoritmo genético casi completo. La única función que como mínimo debe definir el usuario es la función objetivo o función fitness. Esta función obtiene una medida que indica la bondad de un individuo frente al resto. Dicha medida es la utilizada por el operador de selección para discriminar entre individuos.

La función objetivo que el usuario debe definir toma como argumento la referencia del individuo a evaluar y devuelve un valor que indica cómo de bueno o malo es dicho

individuo. La signatura de esta función debe ser la siguiente:

```
1 float Objective(GAGenome &) {  
2     // Aquí se debe de detallar el código de la función objetivo  
3 }
```

Como se puede ver en este código, la función recibe como argumento un individuo genérico y por lo tanto para poder trabajar con dicho individuo dentro de la función, será necesario llevar a cabo el casting al tipo concreto de individuo que se esté utilizando. Por ejemplo, si el individuo es del tipo binario, la función comenzaría de la siguiente forma:

```
1 float Objective(GAGenome & g) {  
2     // Se realiza el casting correspondiente  
3     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;  
4     ...  
5 }
```

6.6.1 Ejemplos de funciones objetivo

Para clarificar la definición de esta función objetivo vamos a mostrar una función objetivo posible para los dos primeros problemas planteados: minimizar una función y N reinas.

Función objetivo para el problema de minimizar una función

Para el problema de minimizar la función y usando la representación del individuo comentada en la Sección 6.3.3, la medida de la bondad o fitness de un individuo es simplemente el valor Y obtenido al aplicar el punto $X1$ y $X2$ representado en el individuo a la función que estamos minimizando. Sabemos que analíticamente, el valor óptimo lo proporcionará una solución con fitness 1, es decir, $Y = 1$.

El método `float * decodificar(GAGenome & g)` es el descrito en la Sección 6.3.3. Como comentamos, este método se encarga de obtener el fenotipo del individuo g , es decir, los valores decimales $X1$ y $X2$ representados a partir de dicho individuo.

```
1 float Objective(GAGenome & g) {  
2     // Se realiza el casting correspondiente  
3     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
```

```

4
5 // Se devuelve el valor de Y
6 float * vect , X1, X2;
7 vect = decodificar(genome);
8 X1 = vect [0];
9 X2 = vect [1];
10 delete [] vect;
11 float Y =(X1*X1) - (2*X1) + (X2*X2) + 2;
12 return Y;
13 }

```

Función objetivo para el problema de las N reinas

Como segundo ejemplo vamos a mostrar una función objetivo para el problema de las N reinas, partiendo de la representación comentada en la Sección 6.5.4. En este problema, una posible medida de la bondad de una solución o un cromosoma es el número de jaques que se producen en la misma. Por lo tanto, el valor óptimo lo proporcionará una solución con fitness 0, es decir, con un número de jaques igual a 0.

```

1 float Objective(GAGenome & g) {
2 // Se realiza el casting correspondiente
3 GA1DArrayAlleleGenome<int> & genome = (GA1DArrayAlleleGenome<int> &)g;
4 float jaques=0;
5 int c, f;
6
7 //Si hay un repetido en vector es un jaque de misma fila
8 for(int i=0; i<genome.length(); i++)
9     for(int j=i+1; j<genome.length(); j++)
10        if (genome.gene(i)==genome.gene(j)) jaques++;
11
12 //Si diagonal también es jaque
13 for(int en_est=0; en_est<genome.length(); en_est++){
14
15 //Diagonal derecha abajo
16 c=en_est+1;
17 f=genome.gene(en_est)+1;
18 while ((c<genome.length()) && (f<genome.length())){
19     if (genome.gene(c)==f) jaques++;
20     c++;
21     f++;
22 }

```



```
23 //Diagonal derecha arriba
24 c=en_est+1;
25 f=genome.gene(en_est)-1;
26 while ((c<genome.length()) && (f>=0)){
27     if (genome.gene(c)==f) jaques++;
28     c++;
29     f--;
30 }
31 }
32
33 return jaques;
34 }
```


Capítulo 7

Definición del algoritmo genético

7.1 Introducción

La clase que permite crear una instancia de un algoritmo genético es `GAGeneticAlgorithm`. Es una clase abstracta y no puede ser instanciada. Para crear una instancia de un algoritmo genético hay que crear una instancia de las diferentes clases derivadas que representan los diferentes tipos de algoritmos genéticos que la librería `GAlib` ofrece. Estas clases son las siguientes:

- La clase derivada `GASimpleGA` representa el tipo de algoritmo genético más sencillo y lo ampliaremos en los siguientes apartados. Su característica principal es que crea una población totalmente nueva en cada generación.
- La clase `GADemeGA` representa un tipo de algoritmo genético con la particularidad de permitir la ejecución de varias poblaciones paralelas permitiendo la migración de individuos entre ellas.
- El tipo de algoritmo genético representado en la clase `GAIncrementalGA` tiene la particularidad de permitir un pequeño solapamiento de las poblaciones generadas (con individuos comunes) con un reemplazamiento personalizado por el usuario. En cada generación se producen varios hijos (normalmente 1 ó 2) que deben sustituir a otros individuos de la población.
- La clase derivada `GASteadyStateGA` representa un algoritmo que produce poblaciones solapadas, permitiendo especificar qué proporción de la población debe ser

sustituida en cada generación.

7.2 La clase GASimpleGA

Como comentamos anteriormente nos vamos a centrar en el algoritmo genético que está implementado en la clase `GASimpleGA`. Este algoritmo genético es el que se describe en [4]. Las poblaciones en este algoritmo no se superponen, es decir, se crea una población totalmente nueva en cada generación. El esquema básico del funcionamiento de este tipo de algoritmo es el siguiente:

```
1 función Algoritmo Genético
2     t = 0
3     inicializar P(t)
4     evaluar P(t)
5     mientras (no condición de parada) hacer
6         t = t + 1
7         seleccionar P(t) desde P(t-1)
8         cruzar P(t)
9         mutación P(t)
10        evaluar P(t)
```

De esta forma, al crear un algoritmo de este tipo usando la librería `GAlib`, se debe de especificar un individuo o una población de individuos. El algoritmo genético clonará el/los individuos que se hayan especificado para crear la población inicial. En cada generación el algoritmo crea una nueva población de individuos aplicando una selección de la población anterior, aplicando los operadores genéticos (cruce y mutación) para crear la población descendiente de la nueva población. El proceso continúa hasta que se cumple el criterio de terminación (este criterio se determina mediante el objeto `Terminator`).

La librería permite opcionalmente que este tipo de algoritmo sea elitista. El elitismo permite que el mejor individuo de cada generación pase automáticamente a formar parte de la siguiente generación. Por defecto, el elitismo está activado. Para desactivar el elitismo se utiliza el método `GABoolean elitist(GABoolean flag)` indicando como parámetro el valor `gaFalse`.

7.2.1 Constructor de clase

Para crear un algoritmo genético simple, la clase proporciona los tres constructores de clase siguientes:

- `GASimpleGA(const GAGenome &)`: Crea la población inicial de individuos a partir del individuo que se le pasa como parámetro.
- `GASimpleGA(const GAPopulation &)`: Crea la población inicial de individuos copiando la población que se pasa como parámetro.
- `GASimpleGA(const GASimpleGA &)`: Crea el algoritmo genético como copia del que se le pasa como parámetro.

7.2.2 Métodos principales

Además de los constructores de clase, el algoritmo genético dispone de una serie de métodos para establecer los diferentes parámetros del algoritmo. A continuación se detallan los más importantes:

- `void evolve(unsigned int seed=0)`: Este método inicia la evolución del algoritmo genético hasta que se cumpla el criterio de parada. Como parámetro se pasa una semilla para la generación de los valores aleatorios necesarios.
- `const GASTatistics& statistics() const`: El método devuelve una referencia al objeto de estadísticas del algoritmo genético. El objeto de estadísticas mantiene diversa información acerca de la ejecución del algoritmo y será comentado más adelante.
- `int generation() const`: Este método devuelve el número de generación actual.
- `int minimaxi() const / int minimaxi(int)`: Este método devuelve / define si el objetivo del algoritmo genético consiste en minimizar o maximizar. El valor a indicar cuando el problema sea de minimizar será de -1 y para maximizar el valor a indicar como parámetro es 1. Por defecto, el valor establecido es maximizar.

- `int nGenerations() const / int nGenerations(unsigned int)`: El método devuelve / establece el número de generaciones máxima que el algoritmo debe realizar.
- `float pMutation() const / float pMutation(float)`: Este método devuelve / define la probabilidad de mutación.
- `float pCrossover() const / float pCrossover(float)`: Este método devuelve / define la probabilidad de cruce.
- `int populationSize() const / int populationSize(unsigned int)`: Este método devuelve / define el tamaño de la población. Dicho tamaño puede ser modificado durante la evolución del algoritmo genético.
- `GASelectionScheme& selector() const / GASelectionScheme& selector(const GASelectionScheme& s)`: Este método devuelve / define el operador de selección. Los esquemas posibles proporcionados por la librería como clases son:
 - `GARouletteWheel`: Se asigna una probabilidad de selección proporcional al valor del fitness del cromosoma.
 - `GATournamentSelector`: Escoge al individuo de mejor fitness de entre Nts individuos seleccionados aleatoriamente con reemplazamiento ($Nts = 2, 3, \dots$).
 - `GAUniformSelector`: Todos los individuos de la población tienen la misma probabilidad de ser seleccionados.
 - `GARankSelector`: Selecciona al individuo con mejor fitness.

El operador por defecto es `RouletteWheel`. Un ejemplo que cambiaría el operador de selección por defecto de un algoritmo es el siguiente:

```
1 GA TournamentSelector s; // declaramos un objeto del tipo selección por
   torneo
2 ga.selector(s); // se lo asignamos al algoritmo genético ga
```

7.2.3 Terminación del algoritmo genético

Los métodos `GAGeneticAlgorithm::Terminator terminator()` y `GAGeneticAlgorithm::Terminator terminator(GAGeneticAlgorithm::Terminator)` devuelve / define el criterio de terminación del algoritmo genético. El criterio de terminación por defecto en un algoritmo genético del tipo `GASimpleGA` es el número de generaciones.

El usuario puede definir otros criterios de terminación y asignárselos al algoritmo genético usando el método `terminator` comentado anteriormente:

```
1  ...
2  GASimpleGA ga;
3  ga.terminator(FuncionTerminacion);
4  ...
5  GABoolean FuncionTerminacion(GAGeneticAlgorithm &){
6  // código que define el criterio de parada
7  }
```

donde podemos ver que la signatura de una función de terminación debe ser la mostrada en el código anterior: un parámetro de entrada que es el objeto algoritmo genético sobre el que se define la función de terminación y devuelve un valor `GABoolean` (`GATrue` o `GAFalse`) según se cumpla o no el criterio de parada respectivamente.

Por último, a modo de ejemplo mostramos a continuación la parte inicial del problema de minimizar una función. De esta forma veremos cómo se crea el algoritmo, cómo se establecen los parámetros y cómo el algoritmo comienza a evolucionar.

```
1  ...
2  int main(int argc, char **argv){
3
4  // Especificamos una semilla aleatoria.
5  GARandomSeed((unsigned int)atoi(argv[1]));
6
7  // Declaramos variables para los parámetros del GA e inicializamos algunos
   valores
8  int popsize = 200;
9  int ngen = 5000;
10 float pmut = 0.01;
11 float pcross = 0.6;
12 int tam = 32;
13 GA1DBinaryStringGenome genome(tam, Objective, NULL);
```

```
14 GASimpleGA ga(genome);
15 ga.minimaxi(-1); // minimizamos
16 ga.populationSize(popsiz);
17 ga.nGenerations(ngen);
18 ga.pMutation(pmut);
19 ga.pCrossover(pcross);
20 ga.evolve();
21 ...
22 }
23 ...
```

7.3 Impresión de valores de la ejecución

El diseño de la solución de un problema utilizando un algoritmo genético conlleva el estudio y análisis de diversos parámetros. Un aspecto interesante que ofrece la librería `GAlib` es un conjunto de métodos que muestran diferentes estadísticas acerca de una ejecución del algoritmo genético. Este conjunto de métodos se encuentran en la clase `GASStatistics`. Esta clase se encuentra accesible desde todos los tipos de algoritmos genéticos, ya que todos los objetos de tipo algoritmo genético tienen una función llamada `statistics()` que devuelve un objeto de tipo `GASStatistics`. Por lo tanto, se puede consultar las estadísticas de las ejecuciones sea cual sea el tipo de algoritmo genético utilizado para buscar la solución a un problema planteado. Vamos a analizar un poco más en detalle dicha clase.

7.3.1 Objeto `GASStatistics`

El objeto `GASStatistics` contiene información sobre el estado actual de un algoritmo genético. Cada objeto del tipo algoritmo genético en la librería, tiene asociado un objeto `GASStatistics` al que se puede acceder a través del método `statistics()`.

Cuando un objeto de tipo algoritmo genético accede al método `statistics()` (`ga.statistics()`, siendo `ga` un objeto de tipo algoritmo genético), el método devuelve una referencia al objeto `GASStatistics`. Este objeto mantiene siempre actualizada la información como el mejor individuo, el peor, la media y la desviación típica del fitness, etc.

Vamos a detallar cuales son los métodos principales proporcionados por esta clase para mostrar los diferentes valores de un algoritmo genético en ejecución y que puede ser muy útil a la hora de ajustar los diferentes parámetros del mismo.

7.3.2 Métodos principales

Entre los métodos principales para imprimir información de la ejecución de un algoritmo genético encontramos los siguientes:

- `float bestEver() const`: Retorna el fitness del mejor individuo encontrado hasta el momento.
- `float maxEver() const`: Devuelve el fitness máximo desde la inicialización.
- `float minEver() const`: Devuelve el fitness mínimo desde la inicialización.
- `const GAGenome& bestIndividual(unsigned int n=0) const`: Devuelve una referencia al mejor individuo encontrado por el algoritmo genético.
- `int crossovers() const`: Devuelve el número de cruces que se han llevado a cabo desde la inicialización.
- `int generation() const`: Devuelve el número de generación actual.
- `int mutations() const`: Devuelve el número de mutaciones que se han llevado a cabo desde la inicialización de la población.

Un ejemplo del uso de estas estadísticas lo encontramos en la definición de una función de terminación para el problema de las N reinas, donde establecemos que el algoritmo genético terminará al alcanzar el fitness óptimo (0 en este problema) o el número de generaciones especificado:

```
1 // Definimos la función de terminación para el problema de las  $N$  reinas
2
3 GABoolean Termina(GAGeneticAlgorithm & ga){
4     if ((ga.statistics().minEver()==0)|| (ga.statistics().generation()==ga.
5         nGenerations())) return gaTrue;
6     else return gaFalse;
}
```

Pero también podemos utilizar estas estadísticas para estudiar si un determinado operador tiene un funcionamiento correcto mirando el número de veces que el operador ha sido efectivo. Un ejemplo de uso para este fin sería el siguiente:

```

1 int umbral = popsize/1000;
2
3 // Definimos una función de determina si el número de cruces realizados es
  menor que cierto umbral, predefinido anteriormente.
4
5 void OperadorCruceCorrecto(GAGeneticAlgorithm &ga){
6     if ((ga.statistics().crossovers() < umbral))
7         cout << "El operador de cruce realiza pocas operaciones" <<
            endl;
8 }

```

7.4 Código completo para la optimización de una función

Una vez analizados los distintos elementos necesarios para la resolución de este problema usando un algoritmo genético implementado mediante la librería GALib, mostramos el código completo del mismo:

```

1
2 /* -----
3  Optimización de una función
4
5  Programa que ilustra cómo usar un GA para encontrar el valor mínimo de
   una función continua en dos variables. Usa un genoma string binario
6
7  ----- */
8
9 #include <ga/GASimpleGA.h> // usaremos un algoritmo genético simple
10 #include <ga/GA1DBinStrGenome.h> // y el genoma string binario de 1
    dimensión
11 #include <ga/std_stream.h>
12 #include <iostream>
13 #include <math.h>
14
15 using namespace std;
16

```

```
17 float Objective(GAGenome &); // La función objetivo la definimos más
    adelante
18
19 float * decodificar(GAGenome &); // Función que obtiene el fenotipo de un
    genoma
20
21 int main(int argc, char **argv){
22
23     cout << "Este programa encuentra el valor minimo en la funcion\n";
24     cout << " y = x1^2 - 2x^1 + x2^2 + 2\n";
25     cout << "con las restricciones\n";
26     cout << "      0 <= x1 <= 5\n";
27     cout << "      0 <= x2 <= 5\n";
28     cout << "\n\n";
29     cout.flush();
30
31 // Especificamos una semilla aleatoria.
32
33     GARandomSeed((unsigned int)atoi(argv[1]));
34
35 // Declaramos variables para los parámetros del GA e inicializamos algunos
    valores
36
37     int popsize = 200;
38     int ngen = 5000;
39     float pmut = 0.01;
40     float pcross = 0.6;
41
42 // El genoma contiene 16 bits para representar el valor de X1 y 16 bits
    para representar el valor de X2
43
44     int tam = 32;
45
46 // del gen 0 al 15 representa el valor de X1
47 // del gen 16 al 31 representa el valor de X2
48
49 // Ahora creamos el GA y lo ejecutamos. Primero creamos el genoma que es
    usado por el GA para clonarlo y crear una población de genomas.
50
51     GA1DBinaryStringGenome genome(tam, Objective, NULL);
52
53 // Una vez creado el genoma, creamos el algoritmo genético e inicializamos
    sus parámetros - tamaño de la población, número de generaciones,
    probabilidad de mutación, y probabilidad de cruce. Finalmente, le
```

```

    indicamos que evolucione.
54
55   GASimpleGA ga(genome);
56   ga.minimaxi(-1); // minimizamos
57   ga.populationSize(popsiz);
58   ga.nGenerations(nngen);
59   ga.pMutation(pmut);
60   ga.pCrossover(pcross);
61   ga.evolve();
62
63 // recuperamos el mejor individuo y obtenemos su fenotipo
64
65   GA1DBinaryStringGenome & mejor = (GA1DBinaryStringGenome &)ga.statistics
    (.bestIndividual());
66   float * vect = decodificar(mejor);
67
68 // Imprimimos el mejor genoma encontrado por el GA, su fitness y los
    valores X1 y X2
69
70   cout << "El GA encuentra:\n" << ga.statistics().bestIndividual() << "\n\n"
    ";
71   cout << "Mejor valor fitness es y = " << ga.statistics().minEver() << "\n"
    "\n";
72   cout << "El punto encontrado es <" << vect[0] << ", " << vect[1] << ">" <<
    endl;
73
74   delete [] vect;
75
76   system("pause");
77   return 0;
78 }
79
80 // decodifica el genoma
81
82 float * decodificar(GAGenome & g){
83
84   GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
85
86   float * vect = new float [2];
87
88 // del gen 0 al 15 representa el valor de X1
89   float parte1 = 0;
90   for(int i=0; i<16; i++)
91       parte1+=(genome.gene(i) * pow(2.0, (float)15-i));

```

```

92  float X1 = 0 + (parte1/65535.0) * (5-0);
93
94  // del gen 16 al 31 representa el valor de X2
95  float parte2 = 0;
96  for(int i=16; i<32; i++)
97      parte2+=(genome.gene(i) * pow(2.0,(float)31-i));
98  float X2 = 0 + (parte2/65535.0) * (5-0);
99
100 vect[0] = X1;
101 vect[1] = X2;
102
103 return vect;
104 }
105
106 // Definimos la función objetivo — minimizar  $y = X1*X1 - 2*X1 + X2*X2 + 2$ 
107
108 float Objective(GAGenome& g) {
109     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
110
111     // devolvemos el valor de Y
112
113     float * vect, X1, X2;
114
115     vect = decodificar(genome);
116     X1 = vect[0];
117     X2 = vect[1];
118
119     delete [] vect;
120
121     float Y =(X1*X1) - (2*X1) + (X2*X2) + 2;
122
123     return Y;
124 }

```

Cuando ejecutamos el programa obtenemos una salida como la mostrada en la Figura 7.1.

7.5 Código completo del ejemplo de las N reinas

El código completo que resuelve mediante un algoritmo genético el problema de las N reinas es el siguiente:

```

C:\Users\Carmen\Desktop\material_didactico\material\para_raquel\codigo\funcion\funcion_lineal...
Este programa encuentra el valor minimo en la funcion
y = x1^2 - 2x1 + x2^2 + 2
con las restricciones
  0 <= x1 <= 5
  0 <= x2 <= 5

El GA encuentra:
001100110011010000000000000010

Mejor valor fitness es y = 1

El punto encontrado es <1.00008,0.00015259>
Presione una tecla para continuar . . . _

```

Figura 7.1: Salida del problema de optimización de una función

```

1 /* ————— PROBLEMA DE LAS N REINAS ————— */
2
3 #include <ga/GASimpleGA.h> // Algoritmo Genético simple
4 #include <ga/GA1DArrayGenome.h> // genoma —> array de enteros (dim. 1)
5 #include <iostream>
6 #include <fstream>
7 using namespace std;
8
9 float Objective(GAGenome &); // Función objetivo —> definida más adelante
10 GABoolean Termina(GAGeneticAlgorithm &); // Función de terminación —>
    definida más adelante
11
12 int main(int argc, char **argv){
13
14 int nreinas = 8;
15
16 cout << "Problema de las " << nreinas << " reinas \n\n";
17 cout.flush();
18
19 // Establecemos una semilla para el generador de números aleatorios
20
21 GARandomSeed((unsigned int)atoi(argv[1]));
22

```

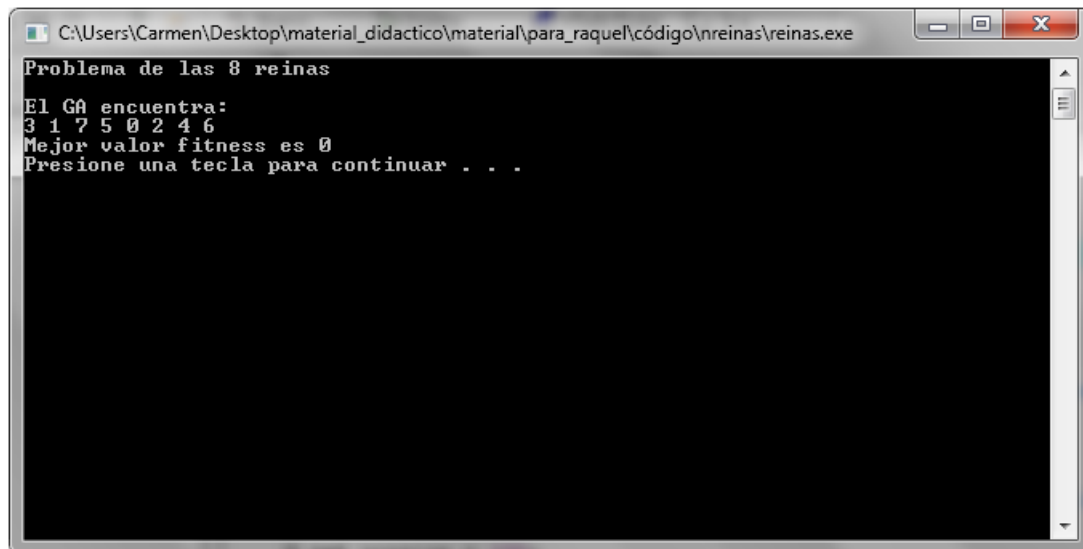
```
23 // Declaramos variables para los parámetros del GA y las inicializamos
24
25 int popsize = 100;
26 int ngen = 10000;
27 float pmut = 0.1;
28 float pcross = 0.9;
29
30 // Conjunto enumerado de alelos —> valores posibles de cada gen del genoma
31
32 GAAlleleSet<int> alelos;
33 for(int i = 0; i < nreinas; i++) alelos.add(i);
34
35 // Creamos el genoma
36
37 GA1DArrayAlleleGenome<int> genome(nreinas , alelos , Objective , NULL);
38
39 // Creamos el algoritmo genético
40
41 GASimpleGA ga(genome);
42
43 // Inicializamos – minimizar la función objetivo , tamaño de la población , n
    . generaciones , prob. mutación , prob. cruce , función de terminación y
    le indicamos que evolucione.
44
45 ga.minimaxi(-1);
46 ga.populationSize(popsize);
47 ga.nGenerations(ngen);
48 ga.pMutation(pmut);
49 ga.pCrossover(pcross);
50 ga.terminator(Termina);
51 ga.evolve();
52
53 // Imprimimos el mejor individuo que encuentra el GA y su valor fitness
54
55 cout << "El GA encuentra:\n" << ga.statistics().bestIndividual() << "\n";
56 cout << "Mejor valor fitness es " << ga.statistics().minEver() << "\n";
57
58 system("pause");
59
60 return 0;
61 }
62
63 // Definimos la función objetivo.
64
```

```

65 float Objective(GAGenome& g) {
66     GA1DArrayAlleleGenome<int> & genome = (GA1DArrayAlleleGenome<int> &)g;
67     float jaques = 0;
68     int c, f;
69
70     // si repetido en vector es un jaque de misma fila
71
72     for(int i = 0; i < genome.length(); i++)
73         for(int j = i+1; j < genome.length(); j++)
74             if (genome.gene(i) == genome.gene(j)) jaques++;
75
76     // si diagonal también es jaque
77
78     for(int en_est = 0; en_est < genome.length(); en_est++){
79
80         // diagonal derecha abajo
81         c = en_est+1;
82         f = genome.gene(en_est)+1;
83         while ((c < genome.length()) && (f < genome.length())){
84             if (genome.gene(c) == f) jaques++;
85             c++;
86             f++;
87         }
88         // diagonal derecha arriba
89         c = en_est + 1;
90         f = genome.gene(en_est) - 1;
91         while ((c < genome.length()) && (f >= 0)){
92             if (genome.gene(c) == f) jaques++;
93             c++;
94             f--;
95         }
96     }
97     return jaques;
98 }
99
100 // Definimos la función de terminación para que termine cuando alcance el
    número de generaciones o el fitness 0
101
102 GABoolean Termina(GAGeneticAlgorithm &ga){
103     if ((ga.statistics().minEver() == 0) || (ga.statistics().
        generation() == ga.nGenerations())) return gaTrue;
104     else return gaFalse;
105 }

```


Cuando ejecutamos el programa obtenemos una salida como la mostrada en la Figura 7.2.



```
C:\Users\Carmen\Desktop\material_didactico\material\para_raquel\codigo\reinas\reinas.exe
Problema de las 8 reinas
El GA encuentra:
3 1 7 5 0 2 4 6
Mejor valor fitness es 0
Presione una tecla para continuar . . .
```

Figura 7.2: Salida del problema de las N reinas

7.6 Resolviendo el problema de la mochila 0-1

Por último, abordamos la resolución del problema de la mochila 0-1.

Para resolver este problema, usaremos un individuo formado por un string binario de dimensión 1 y de tamaño igual al número de objetos disponibles en el problema y susceptibles de ser introducidos en la mochila. De esta forma, el gen i tomará el valor 1 cuando el objeto i está dentro de la mochila y tomará valor 0 cuando el objeto i no está en la mochila.

El fitness asignado a un individuo será la suma de los beneficios de los objetos introducidos en la mochila en dicha solución (con gen a 1 en la representación). Por lo tanto, el algoritmo genético tratará de encontrar el individuo con mayor fitness (mayor beneficio) que no sobrepase la capacidad o peso máximo de la mochila determinado por el problema. Es debido a esta última restricción (la capacidad o peso máximo de la mochila) por lo que no podremos usar los operadores genéticos proporcionados por la librería y tendremos que definir unos nuevos. Los operadores definidos (inicio, cruce

y mutación) deberán contemplar que los nuevos individuos obtenidos no sobrepasan la capacidad máxima de la mochila ya que en este caso los individuos no son válidos. Por lo tanto, los operadores que definimos proporcionarán siempre individuos que respetan la restricción.

```

1  /* -----
2
3   Los datos del problema se leen de un fichero
4
5   Se definen el inicializador, el cruce y la mutación para que no generen
6   individuos no válidos
7
8   Se ejecuta mochila fichero_datos semilla
9
10 -----*/
11
12 #include <ga/GASimpleGA.h> // usaremos un algoritmo genético simple
13 #include <ga/GA1DBinStrGenome.h> // y el genoma string binario de 1
    dimensión
14 #include <ga/std_stream.h>
15 #include <iostream>
16 #include <fstream>
17 using namespace std;
18
19 float Objective(GAGenome &); // La función objetivo la definimos más
    adelante
20
21 void miInicio(GAGenome &); // Definimos un operador de inicio
22
23 int miMutacion(GAGenome &, float); // Definimos un operador de mutación
24
25 int miCruce(const GAGenome&, const GAGenome &, GAGenome*, GAGenome*); //
    Definimos un operador de cruce
26
27 struct datos{
28     int capacidad; // capacidad de la mochila
29     int * pebe[2]; // pebe[0][i] contiene el peso del objeto i y pebe
        [1][i] contiene el beneficio del objeto i
30 };
31
32 int main(int argc, char **argv)
33 {
34     cout << "Mochila 0/1 \n\n";

```

```
35 cout << "Este programa trata de resolver el problema de la mochila\n\n";
36 cout.flush();
37
38 // Especificamos una semilla aleatoria.
39
40 GARandomSeed((unsigned int)atoi(argv[2]));
41
42 // Declaramos variables para los parametros del GA e inicializamos algunos
   valores
43
44 int popsize = 100;
45 int ngen = 2000;
46 float pmut = 0.001;
47 float pcross = 0.9;
48
49 // Leemos datos del problema
50
51 struct datos D;
52 int tam;
53
54 ifstream f(argv[1]);
55
56 f>>D.capacidad; // se lee la capacidad de la mochila
57 f>>tam; // se lee el número de objetos
58
59 D.pebe[0]=new int[tam];
60 D.pebe[1]=new int[tam];
61
62 for(int kk=0;kk<tam;kk++) {
63     f>>D.pebe[1][kk]; // se leen beneficios
64     f>>D.pebe[0][kk]; // se leen pesos
65 }
66
67 f.close();
68
69 // Ahora creamos el GA y lo ejecutamos. Primero creamos el genoma que es
   usado por el GA para clonarlo y crear una población de genomas.
70
71 // Especificamos la función de inicio, de cruce y de mutación que queremos
   utilizar.
72
73 GA1DBinaryStringGenome genome(tam, Objective,(void *)&D);
74 genome.initializer(miInicio);
75 genome.crossover(miCruce);
```

```
76 genome.mutator(miMutacion);
77
78 // Una vez creado el genoma, creamos el algoritmo genético e inicializamos
    sus parámetros: tamaño de la población, número de generaciones,
    probabilidad de mutación, y probabilidad de cruce. Finalmente, le
    indicamos que evolucione.
79
80 GASimpleGA ga(genome);
81 ga.populationSize(popsiz);
82 ga.nGenerations(nngen);
83 ga.pMutation(pmut);
84 ga.pCrossover(pcross);
85 ga.evolve();
86
87 // calculamos la capacidad consumida por la mejor solución encontrada
88
89 float capa=0;
90
91 GA1DBinaryStringGenome & mejor = (GA1DBinaryStringGenome &)ga.statistics
    ().bestIndividual();
92
93 for(int k=0;k<mejor.length();k++)
94     capa+=(mejor.gene(k)*D.pebe[0][k]);
95
96 // Imprimimos el mejor genoma encontrado por el GA, su fitness y capacidad
    usada
97
98 cout << "El GA encuentra:\n" << ga.statistics().bestIndividual() << "\n\n"
    ";
99 cout << "Mejor valor fitness es " << ga.statistics().maxEver() << "\n\n";
100 cout << "Capacidad consumida " << capa << "\n\n";
101
102 system("pause");
103 return 0;
104 }
105
106 // Definimos la función objetivo
107
108 float Objective(GAGenome& g) {
109     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
110     struct datos *D1=(struct datos *)genome.userData(); // recuperamos la
        estructura de datos
111     float valor=0.0;
112
```

```
113 // devolvemos el beneficio
114
115 for(int i=0; i<genome.length(); i++)
116     valor+=(genome.gene(i)*D1->pebe[1][i]);
117
118 return valor;
119 }
120
121 // Definimos una función de inicio que genere individuos válidos
122
123 void miInicio(GAGenome &g){
124     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
125     struct datos *D1=(struct datos *)genome.userData();
126     float valor=0.0;
127
128     for(int i=0;i<genome.length();i++){
129         valor+=D1->pebe[0][i];
130
131         // comprobamos si meter el objeto i en la mochila sobrepasa la
132         // capacidad si la sobrepasa, el objeto no se puede incluir;
133         // si no se sobrepasa, se dedide aleatoriamente si se
134         // incluye
135
136         if (valor>D1->capacidad){
137             genome.gene(i,0);
138             valor-=D1->pebe[0][i];
139         }
140         else {
141             genome.gene(i, GARandomInt());
142             if (!(genome.gene(i))) valor-=D1->pebe[0][i];
143         }
144     }
145 }
146
147 // Definimos un operador de mutación que genere individuos válidos
148
149 // Parámetros de entrada: genoma y probabilidad de mutación
150
151 // Valor de salida: número de mutaciones
152
153 int miMutacion(GAGenome &g, float pmut){
154     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
155     struct datos *D1=(struct datos *)genome.userData();
156     float count=0.0;
```

```

154     int nmut=0;
155
156     if (pmut<=0.0) return 0;
157
158     // calculamos el peso actual de la mochila
159
160     for(int i=0; i<genome.length(); i++)
161         count+=(genome.gene(i)*D1->pebe[0][i]);
162
163     // si un gen debe mutarse, si es un objeto que se extrae no se
164     // comprueba nada, si es un objeto que se incluye se comprueba que no
165     // sobrepase la capacidad de la mochila. Si la sobrepasa el gen no se
166     // muta
167
168     for(int i=0; i<genome.length(); i++){
169         if (GAFlipCoin(pmut)) {
170             nmut++;
171             if (genome.gene(i) genome.gene(i,0);
172             else {
173                 count+=D1->pebe[0][i];
174                 if(count>D1->capacidad) {
175                     count-=D1->pebe[0][i];
176                     nmut--;
177                 }
178                 else genome.gene(i,1);
179             }
180         }
181     }
182     return nmut;
183 }
184
185 // Definimos el operador de cruce para que genere individuos válidos
186 // Parámetros de entrada: los dos genomas padres y los dos hijos
187 // Valor de salida: el numero de hijos que genera
188 int miCruce(const GAGenome& p1,const GAGenome & p2,GAGenome* c1,GAGenome*
189 c2){
190     GA1DBinaryStringGenome & m = (GA1DBinaryStringGenome &)p1;
191     GA1DBinaryStringGenome & p = (GA1DBinaryStringGenome &)p2;
192     struct datos *D1=(struct datos *)m.userData();
193     int n=0;

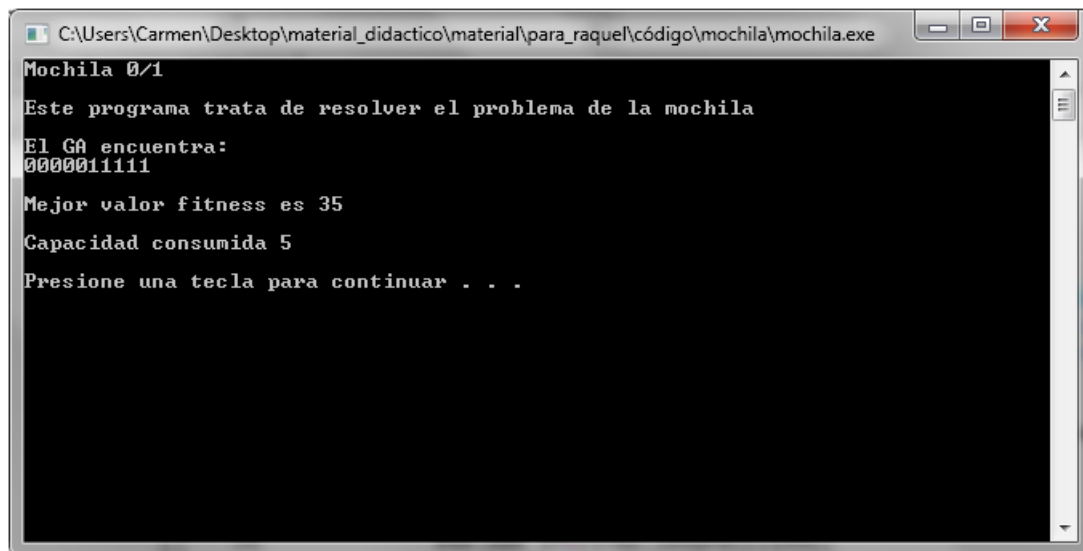
```

```
194     int punto1=GARandomInt(0,m.length()); // generamos el punto de cruce
195     int punto2=m.length()-punto1;
196
197     // el hijo 1 hereda la primera parte del padre 1 y la segunda parte del
        // padre 2. Comprobamos el peso del hijo generado y cuando un objeto
        // hace que se sobrepase la capacidad de la mochila, se extrae.
198
199     if (c1){
200         GA1DBinaryStringGenome & h1= (GA1DBinaryStringGenome &)*c1;
201         float count=0.0;
202         h1.copy(m,0,0,punto1);
203         h1.copy(p,punto1,punto1,punto2);
204         for(int i=0;i<h1.length();i++){
205             count+=(h1.gene(i)*D1->pebe[0][i]);
206             if (count>D1->capacidad) {
207                 h1.gene(i,0);
208                 count-=D1->pebe[0][i];
209             }
210         }
211         n++;
212     }
213
214     // el hijo 2 hereda la primera parte del padre 2 y la segunda parte del
        // padre 1. Comprobamos el peso del hijo generado y cuando un objeto
        // hace que se sobrepase la capacidad de la mochila, se extrae.
215
216     if (c2){
217         GA1DBinaryStringGenome & h2= (GA1DBinaryStringGenome &)*c2;
218         float count=0.0;
219         h2.copy(p,0,0,punto1);
220         h2.copy(m,punto1,punto1,punto2);
221         for(int i=0;i<h2.length();i++){
222             count+=(h2.gene(i)*D1->pebe[0][i]);
223             if (count>D1->capacidad) {
224                 h2.gene(i,0);
225                 count-=D1->pebe[0][i];
226             }
227         }
228         n++;
229     }
230     return n;
231 }
```

Cuando ejecutamos el programa usando como datos los mostrados en la Tabla 7.1, obtenemos una salida como la mostrada en la Figura 7.3.

5
10
0 1
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
capacidad
número de objetos
beneficio y peso de cada objeto

Tabla 7.1: Datos para una instancia de la mochila 0-1



```
C:\Users\Carmen\Desktop\material_didactico\material\para_raquel\codigo\mochila\mochila.exe
Mochila 0/1
Este programa trata de resolver el problema de la mochila
El GA encuentra:
0000011111
Mejor valor fitness es 35
Capacidad consumida 5
Presione una tecla para continuar . . .
```

Figura 7.3: Salida del problema de la mochila 0-1

Parte III

BIBLIOGRAFÍA

Bibliografía

- [1] M. Wall. GALib – A C++ Library of Genetic Algorithm Components. Massachusetts Institute of Technology (MIT). Documentación oficial de GALib: “<http://lancet.mit.edu/ga/>”.
- [2] C++ con Clase. Documentación de C++: “<http://c.conclase.net/>”
- [3] L. J. Aguilar, L. S. García. Programación en C++: un enfoque práctico. McGraw-Hill, 2006.
- [4] D. E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.
- [5] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. Numerical Recipes in C. Cambridge University Press, 1992.
- [6] W. Savitch. Resolución de problemas con C++: El objetivo de la programación. Prentice Hall, 2000.

