UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Programación de videojuegos I

Unidad 1
Manejo de arreglos

Docentes
Germán Gaona
Mauro Walter

CONTENIDOS

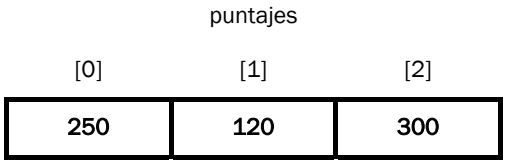
INTRODUCCIÓN

En esta unidad haremos hincapié en los arreglos, una de las estructuras de datos más utilizadas en el mundo de la programación; repasaremos algunos conceptos dados en "Introducción a la programación"; veremos con más detalle temas importantes, como los tipos de búsqueda existentes, y analizaremos en profundidad los ordenamientos más comunes.

Un *arreglo* es una colección de variables del mismo tipo a la que se hace referencia por medio de un nombre común. Los arreglos pueden tener una o varias dimensiones, aunque el arreglo de una dimensión es el más común. En general, se denomina *matriz* cuando el arreglo posee dos dimensiones.

Los arreglos adquieren una importancia aún mayor en el lenguaje C++, pues las cadenas de texto son almacenadas como arreglos del tipo carácter (char), ya que no existe un tipo intrínseco de datos string. Este enfoque confiere mayor flexibilidad en el manejo de cadenas, aunque a costa de cierta complejidad. Cabe aclarar, además, que los arreglos tienen una relación muy estrecha con los punteros y pueden ser suplantados por ellos en muchas situaciones.

La siguiente figura muestra un arreglo unidimensional, que está compuesto por una serie de elementos ubicados de manera contigua y a los que se accede individualmente a través del nombre y un índice. En esta oportunidad, el arreglo contiene los puntajes de tres jugadas sucesivas:



A modo de repaso, se incluye un fragmento de código en C++, ejemplificando la declaración e inicialización del arreglo *puntajes,* para luego recorrer secuencialmente el arreglo, sumando cada uno de ellos y obteniendo, finalmente, el promedio.

```
int main()
{
        //Declaración e inicialización de
        //arreglo unidimensional
        int puntajes [] = {250, 120, 300};

        int i, promedio=0;

        //Recorrido y suma de cada elemento
        for ( i=0 ; i<3 ; i++ )
        {
                promedio += puntajes[i];
        }

        promedio /=i;

        cout << promedio;
        return 0;
}
```

En el caso de arreglos con una dimensión mayor a 1, debemos recordar que C++ tiene una sintaxis diferente a la mayoría de los lenguajes, ya que su declaración de asemeja a un arreglo de arreglos.

Uno de los usos más frecuentes de arreglos multidimensionales es el de representar tableros de juegos como matrices. Tomemos como ejemplo el conocido juego Tatetí, que consiste en un tablero de 3x3, en el que dos contrincantes se enfrentan y sale victorioso aquel que logra llenar primero tres casilleros contiguos, ya sea en forma

horizontal, vertical o diagonal. La siguiente matriz representa el tablero en una posición claramente ganadora para *X:*

tablero

	[0]	[1]	[2]
[0]	X	O	
[1]	X	O	
[2]	X		

La función *verificar* comprueba si alguno de los jugadores ha alcanzado la condición de ganador. Esto debe hacerse cada vez que un jugador realiza un movimiento. Prestemos especial atención a la matriz que se pasa como parámetro y a la forma en que se declara. Como mencionamos anteriormente, en lugar de utilizar comas para separar dimensiones, como en la mayoría de los lenguajes, recurrimos a corchetes por separado.

```
char verificar(int tablero[3][3])
{
  //Verifica filas
  for(int i=0; i<3; i++)
    if(tablero[i][0]==tablero[i][1] &&
       tablero[i][0]==tablero[i][2])
           return tablero[i][0];

  //Verifica columnas
  for(int i=0; i<3; i++)
    if(tablero[0][i]==tablero[1][i] &&
       tablero[0][i]==tablero[2][i])
           return tablero[0][i];

  //Verifica diagonales
  if(tablero[0][0]==tablero[1][1] &&
     tablero[1][1]==tablero[2][2])
         return tablero[0][0];

  if(tablero[0][2]==tablero[1][1] &&
     tablero[1][1]==tablero[2][0])
         return tablero[0][2];

  return ' ';
}
```

Algunas características de los arreglos en C++

- Almacenados en posiciones contiguas de memoria.
- El compilador no realiza chequeo de desbordes por cuestiones de performance.
- En general, son intercambiables con los punteros, siendo estos últimos más eficientes.
- La copia directa entre arreglos no está permitida, debe hacerse elemento a elemento.
- Pueden crearse sin definición de longitud inicial y determinarse al momento de la ejecución.

1.1. Ordenamiento de arreglos

Hasta el momento hemos hablado de los arreglos centrados en su significado, sus operaciones básicas y características más importantes. Por ello, hemos usado ejemplos que cuentan con pocos elementos, pero sabemos que estas estructuras de memoria pueden crecer considerablemente, llegando a contener cientos, miles o millones de elementos.

Como cualquier situación de la vida en la que manejamos conjuntos de elementos, pero que a la vez nos interesa individualizarlos, necesitamos mantener cierto orden para hacer el trabajo más sencillo. Esto puede apreciarse, por ejemplo, al tomar asistencia en un curso o al ubicar inicialmente las piezas en un tablero de ajedrez.

El ordenamiento de arreglos adquiere real importancia para su uso eficiente, pues la mayoría de las operaciones sobre los elementos contenidos dentro de esta estructura requieren búsquedas previas sobre ellos. Por lo tanto, si un arreglo está ordenado, podríamos aplicar estrategias para buscar el elemento deseado de manera más rápida. Por el contrario, si está desordenado, deberemos hacerlo secuencialmente, recorriendo el arreglo completo, en el peor de los casos.

Sin embargo, la ventaja que confiere el ordenamiento conlleva un punto en contra, que es el tiempo necesario para completar este proceso inicialmente. Es así que diversos algoritmos de ordenamiento que han sido propuestos, en la mayoría de los casos, fueron enfocados en disminuir el tiempo de procesamiento al mínimo, el cual es medido por un indicador conocido como *complejidad algorítmica,* calculado en base a la cantidad de elementos de un arreglo. La complejidad algorítmica, junto a otros indicadores como la cantidad de memoria requerida o la estabilidad, son usados para comparar los diferentes métodos de búsqueda existentes.

A continuación, analizaremos algunos algoritmos más conocidos, todos ellos aplicados sobre arreglos unidimensionales, en los que generalmente se suele concentrar su uso, aunque nada quita que puedan ser adaptados a versiones multidimensionales, si fuese necesario.

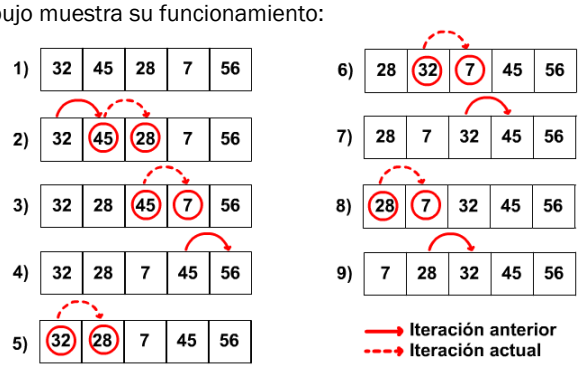
1.1.1. Ordenamiento burbuja

También conocido como el método del intercambio directo, su funcionamiento consiste en revisar cada elemento del arreglo con el siguiente, realizando un cambio de posición a la vez, si fuese necesario. De esta manera, los elementos suben o bajan como si fueran *burbujas.* El algoritmo continúa iterando sobre el arreglo varias veces hasta que la lista quede ordenada.

Su principal ventaja radica en su sencillez de implementación, teniendo en su contra el tiempo que toma ordenar arreglos grandes, debido a la cantidad de comparaciones e intercambios realizados.

Ejemplo:

El siguiente dibujo muestra su funcionamiento:



1. Arreglo desordenado inicial con 5 elementos.

2. Comienza el bucle principal iterando desde la posición 1, hasta que encuentre un elemento menor que el de la posición anterior.

3. Intercambia posición entre el segundo y el tercer elemento y nuevamente se encuentra un elemento menor que la anterior posición.

4. Se realiza el intercambio y se sigue iterando hasta el final del arreglo para verificar si hay un elemento aún menor.

5. Una vez terminado el recorrido del arreglo, se comienza nuevamente desde la primera posición, donde se realiza un intercambio.

6. Intercambio.

7. Aquí vemos cómo el último elemento a comparar es el anteúltimo del arreglo, ya que el último 56 ya está en su posición final.

8. El bucle principal sigue iterando y se realiza un intercambio.

9. Se llega al último elemento de comparación, donde 45 ya tiene su ubicación final.

El último paso del algoritmo es la comparación entre 7 y 28, pero como la condición de menor no se cumple, el arreglo del paso 9 es el arreglo final ordenado.

Ejemplo C++:

```
void burbuja(int arr[], int n){
        int i,j,min;
        for(i=0;i<n-1;i++){ // bucle principal
                for(j=0;j<n-i;j++){ // bucle de comparación
                        if (arr[j]> arr[j+1]){
                                min=arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = min;
                        }
                }
        }
}
```

1.1.2. Ordenamiento por selección directa

Consiste en buscar o *seleccionar* el elemento más pequeño de todo el arreglo, en caso de que se trate de un ordenamiento ascendente (o el más grande de todos, en caso contrario), y se coloca en su posición correspondiente. Este proceso continúa hasta que todos los elementos del arreglo queden ordenados de menor a mayor (o en el segundo caso, de mayor a menor).

Es decir, para la primera iteración tendremos los siguientes pasos:

1. Iterar y seleccionar el menor elemento del arreglo.
2. Intercambiar dicho elemento con el primero.

Para la segunda iteración:

1. Iterar y seleccionar el siguiente menor elemento del arreglo.
2. Intercambiar dicho elemento con el segundo.

Y así sucesivamente.

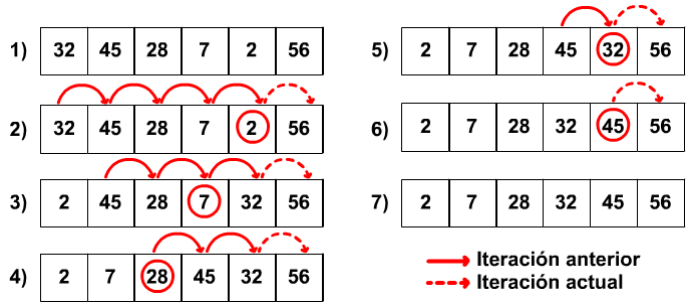
Ejemplo:

1. Tomamos un arreglo con siete elementos, cuyo estado inicial es desordenado. El algoritmo cuenta con dos bucles, el principal y el secundario. El bucle principal es el que dirige todo el proceso; su objetivo es recorrer todo el arreglo secuencialmente de a un elemento a la vez. El bucle secundario

trabaja dentro de los límites impuestos por el principal, tratando de encontrar el menor elemento para intercambiar.

2. Comienza el bucle principal iterando desde la posición 1 hasta que encuentre el elemento menor del arreglo, siendo éste el 2.
3. Intercambia elementos del paso anterior. Ahora, el bucle principal se sitúa en la posición 2 y desde allí comienza el bucle secundario, encontrando a 7 como menor elemento del arreglo.
4. Nuevamente intercambia elementos. El bucle principal se encuentra en la posición 3 y se puede notar que el menor encontrado, 28, se encuentra en la misma posición.
5. Aquí no se realiza intercambio, ya que en realidad el cambio sería la posición 3 en sí misma. Luego, se encuentra 32 como elemento menor.
6. Se realiza el intercambio de elementos, ahora el bucle principal está en la posición 5 y el menor es 45.
7. Se realiza el intercambio y puede verse cómo el arreglo queda completamente ordenado, en este caso, de menor a mayor.

La siguiente figura muestra su funcionamiento:



Ejemplo C++:

```
void seleccionDirecta(int arr[] , int n){{
        int i,j,min,k; bool cambio;
        for(i=0;i<n;i++){ // bucle principal
                min=arr[i];
                k=0;
                cambio=0;
                for(j=i+1;j<n;j++){ // bucle de comparación
                        if (arr[j]<min){
                                min=arr[j];
                                k=j;
                                cambio = 1;
                        }
                }
                if (cambio){
                        arr[k]=arr[i];
                        arr[i]=min;
                }
        }
}
```

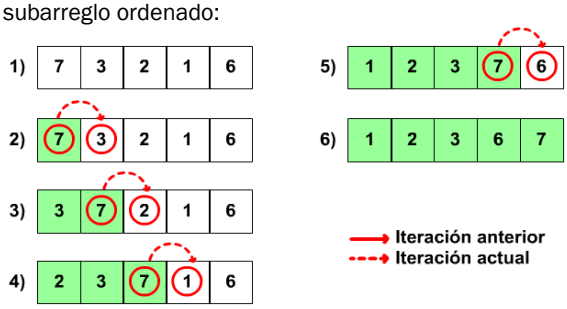
1.1.3. Ordenamiento por inserción

La idea principal de este algoritmo es ordenar un arreglo de entrada en forma incremental, por lo que siempre existe un segmento ordenado y otro desordenado. Su ventaja principal es la simplicidad de implementación y su rapidez para conjuntos de datos pequeños. En contraposición, si se trata de conjuntos grandes, puede tornarse ineficiente.

El algoritmo recibe como dato de entrada el arreglo a ordenar. En la mayoría de las implementaciones se utiliza el mismo arreglo para ir posicionando los elementos en orden, aunque se debe contar con una posición extra de memoria para efectuar los desplazamientos.

Ejemplo:

El siguiente dibujo muestra su funcionamiento, donde las casillas verdes indican cómo va quedando el subarreglo ordenado:



Ejemplo C++:

Existen diversas implementaciones del algoritmo. Aquí presentamos una de las más simples:

```
void insercion(int arr[], int n)
{
  int cur,i;

  // Bucle principal
  for(int j=1;j<n;j++)
  {
     cur=arr[j];
     i=j-1;
     // Bucle de desplazamiento
     while(arr[i]>cur && i>=0)
     {
         arr[i+1]=arr[i];
         i--;
     }
     // Inserción
     arr[i+1]=cur;
  }
}
```

Vemos que el código consta de dos estructuras iterativas anidadas. La más externa implementada con *for* se encarga de recorrer todo el arreglo, desde el segundo elemento hasta el último, de a una posición a la vez. Por su parte, la más interna, implementada con *while,* desplaza los elementos hasta la posición correcta, dejando lugar para el elemento en el orden indicado.

La prueba de escritorio arrojó los siguientes resultados:

Nivel 0	Iter 1	Iter 2
arr=[7,3,2,1,6]; len=5		
	j=1; cur=3	
		i=0; arr[i]=7 arr=[7,7,2,1,6]
	arr=[3,7,2,1,6]	
	j=2; cur=2	
		i=1; arr[i]=7 arr=[3,7,7,1,6] i=0; arr[i]=3 arr=[3,3,7,1,6]
	arr=[2,3,7,1,6]	
	j=3; cur=1	
		i=2; arr[i]=7 arr[2,3,7,7,6,] i=1; arr[i]=3 arr[2,3,3,7,6,] i=0; arr[i]=2 arr[2,2,3,7,6]
	arr=[1,2,3,7,6]	
	j=4; cur=6	
		i=3; arr[i]=7 arr[1,2,3,7,7]
	arr=[1,2,3,6,7]	
arr=[1,2,3,6,7]		

Como podemos apreciar, al final de la iteración más externa, un segmento del arreglo siempre va quedando ordenado:

arr=[7,3,2,1,6]		
	arr=[3,7,2,1,6]	
	arr=[2,3,7,1,6]	
	arr=[1,2,3,7,6]	
	arr=[1,2,3,6,7]	
arr=[1,2,3,6,7]		

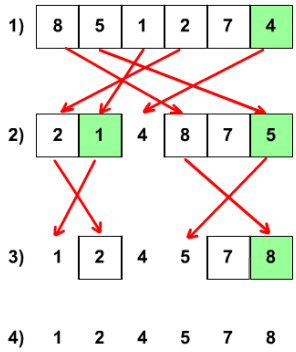
1.1.4. Ordenamiento rápido

Este tipo de ordenamiento se basa en la idea de partición del conjunto de datos (divide y vencerás), siendo su desempeño especialmente bueno para grandes volúmenes. Se podría decir que se divide en tres etapas: elección de un pívot, ordenamiento grueso relativo al pivot y ordenamiento final recursivo. Un factor clave en el comportamiento del algoritmo está relacionado con la correcta elección del pívot y es allí donde la mayoría de los algoritmos se diferencian.

Examinaremos aquí una versión simplificada del algoritmo para estudiar su comportamiento básico, teniendo en cuenta los tres pasos mencionados.

Ejemplo:

El siguiente dibujo muestra su funcionamiento, donde las casillas verdes van señalando el pivote para cada subarreglo:



Ejemplo C++:

```
void quicksort (int arr[], int inf, int sup)
{
        int aux ;
        int i = inf - 1, j = sup;
        int flag = 1;

        // Definición del pivot
        int pivot = arr[sup];

        // Condición de fin
        if (inf >= sup)
                return;

        while (flag)
        {
                while (arr[++i] < pivot);
                while (arr[--j] > pivot);

                // ¨Índices encontrados
                if (i < j)
                {
                        aux = arr[i];
                        arr[i] = arr[j];
                        arr[j] = aux;
                }
                else
                        flag = 0;
        }

        // Ubicamos el pivot
        aux = arr[i];
        arr[i] = arr[sup];
        arr[sup] = aux;

        // Aplicamos ordenamiento recursivamente
        quicksort (arr, inf, i - 1);
        quicksort (arr, i + 1, sup);
}
```

La función de ordenamiento toma tres parámetros: el arreglo a ordenar, el límite inferior y el límite superior. En una de las primeras líneas del código podemos reconocer el establecimiento del pivote. Para concentrarnos en el ordenamiento, aquí se elige siempre el último elemento del segmento.

El algoritmo teórico enuncia que se debe ubicar el pivote en un lugar tal que los elementos a un lado de éste sean menores y al otro, mayores. Para lograrlo, se recorrerá el arreglo desde ambos extremos, usando dos índices distintos. En cada paso, se controla si los elementos opuestos están ubicados en el lugar correcto respecto del pivote. De lo contrario, se intercambian los valores de modo iterativo hasta que sendos índices se encuentren En esta posición se establece el pivote.

Se toma, entonces, cada segmento del arreglo y se aplica ordenamiento de manera recursiva, hasta que se cumpla con la condición de fin, es decir, hasta que el arreglo contenga sólo un elemento. Para decirlo de otra manera, hasta que los límites inferior y superior sean iguales.

A continuación, se detalla la prueba de escritorio para el algoritmo propuesto:

<table>
<tr><td colspan="4">//arreglo desordenado
[8,5,1,2,7,4]</td></tr>
<tr><td colspan="4">pivot=4

i=0; j=3
[2,5,1,8,7,4]

i=1; j=2
[2,1,5,8,7,4]

i=2; j=1
//sin intercambios

//posiciona pivote
[2,1,4,8,7,5]</td></tr>
<tr><td colspan="2">[2,1]</td><td colspan="2">[8,7,5]</td></tr>
<tr><td colspan="2">pivot=1

i=0; j=-1
//sin intercambios

//posiciona pivote
[1,2]</td><td colspan="2">pivot=5

i=3; j=2
//sin intercambios

//posiciona pivote
[5,7,8]</td></tr>
<tr><td>-</td><td>-</td><td>-</td><td>[7,8]</td></tr>
<tr><td></td><td></td><td></td><td>pivot=8
i=5; j=4
//sin intercambios

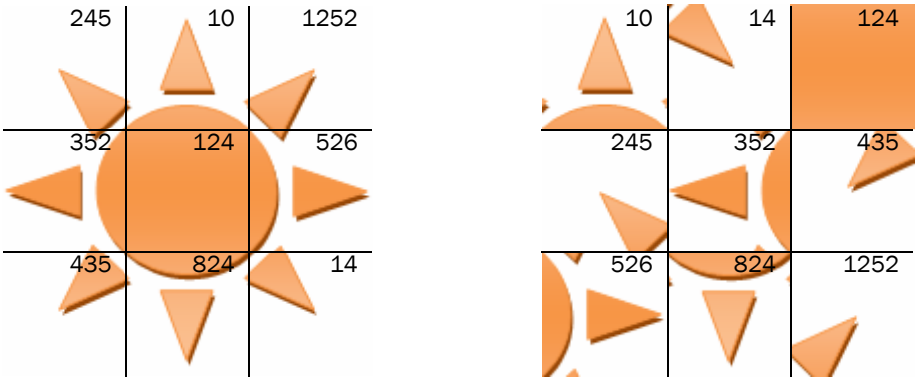
//posiciona pivote
[7,8]</td></tr>
<tr><td></td><td></td><td></td><td>-</td></tr>
<tr><td colspan="4">//arreglo ordenado
[1,2,4,5,7,8]</td></tr>
</table>

Ejemplo de aplicación de ordenamiento

Una vez estudiados los distintos métodos de ordenamiento, su aplicación es sencilla. De hecho, muchos de estos algoritmos se encuentran implementados en la biblioteca de C++.

Imagínense un simple rompecabezas de 3x3 en el que empezamos a jugar con todas las piezas puestas pero desordenadas, con el objetivo final de armar la imagen correcta en el menor tiempo. Para hacer más divertido el juego, deberemos establecer

una configuración inicial distinta cada vez que el mismo reinicia. Una posible solución es hacerlo aleatoriamente, utilizando para ello un arreglo de objetos:



Como se aprecia en la figura, comenzamos con una matriz en la que las piezas están ubicadas correctamente. Luego, debemos preservar estas posiciones en un atributo de cada pieza. Después, para cada uno de los elementos, calculamos un número aleatorio. Ahora, ordenamos las piezas de acuerdo con dichos números y esto nos lleva a tener un rompecabezas inicial desordenado, que era el efecto buscado.

A continuación, presentamos la solución en C++, centrándonos solamente en la parte relacionada con el ordenamiento por simplicidad:

```
#include <iostream>
#include <algorithm>
using namespace std;

//Declaramos la clase Pieza de rompecabezas
class Pieza {
        int fooImg; //Representa la imagen de la pieza
        int initpos; //Representa posición aleatoria relativa
public:
        void setInitPos(int j) {
                initpos=j;
        }
        int getInitPos() {
                return initpos;
        }
        void setFooImg(int j) {
                fooImg=j;
        }
        int getFooImg() {
                return fooImg;
        }
        friend bool operator<(Pieza &a, Pieza &b);
};
//Sobrecarga de operador de comparación
bool operator<( Pieza &a,  Pieza &b)  {
    return (a.initpos < b.initpos);
}

int main()
{
        //Declaramos el rompecabezas como un arreglo
        //de 9 piezas (3x3)
        Pieza rompecabezas[9];
        int i;
        for(i=0; i<9; i++){
                rompecabezas[i].setFooImg(i);
                rompecabezas[i].setInitPos(rand());
        }
        //Usamos una función de ordenamiento
        //de la biblioteca estándar
        sort(rompecabezas,rompecabezas+9);

        //Iteramos para comprobar el ordenamiento
        for(i=0; i<9; i++){
                cout << rompecabezas[i].getFooImg() << "\n";
                cout << rompecabezas[i].getInitPos() << "\n";
        }

        return 0;
}
```

1.2. Búsquedas en arreglos

Los algoritmos de búsqueda ocupan un lugar importante en el manejo de arreglos, pues sin ellos los arreglos se convertirían en meros repositorios de datos.

Estos algoritmos tienen como finalidad primera determinar si el elemento deseado se encuentra en el conjunto en el cual se está buscando; y si el elemento es encontrado, devolver su posición relativa. En general, la estrategia a utilizar para la búsqueda está relacionada con el estado de ordenamiento del arreglo.

Existen tres tipos de búsqueda diferentes:

- Búsqueda secuencial.
- Búsqueda binaria o dicotómica.
- Búsqueda mediante claves (Hashing).

1.2.1. Búsqueda secuencial

Consiste en recorrer y examinar cada uno de los elementos del arreglo hasta encontrar el o los elementos buscados, o hasta que se llegue el final del arreglo. Tiene como ventaja principal su extrema sencillez y el hecho de que el arreglo no necesita estar ordenado. No obstante, lo desfavorece su ineficiente desempeño para grandes conjuntos de datos.

Ejemplo C++:

```
//largo: cantidad de elementos del arreglo.
//arreglo: arreglo de elementos a recorrer.
//elemento: elemento a buscar.
//resultado: arreglo de elementos encontrados.

for(i=j=0;i<largo;i++){
        if(arreglo[i] == elemento){
                resultado[j]=i;
                j++;
        }
}
```

Este algoritmo puede optimizarse cuando el arreglo está ordenado. En este caso, la condición sería:

```
for(i=j=0;arreglo[i]<=elemento;i++)
```

Si se necesita sólo la primera ocurrencia del elemento en el arreglo ordenado, puede utilizarse el siguiente algoritmo:

```
for(i=0;i<largo;i++)
   if(arreglo[i] == elemento)
      break;
```

En este último ejemplo, cuando sólo interesa la primera posición, se puede dar a la posición siguiente al último elemento de arreglo el valor del elemento a buscar. Entonces, de esta manera, estamos seguros de que se encuentra el elemento y no tenemos que comprobar a cada paso si seguimos buscando dentro de los límites del arreglo.

```
arreglo[largo+1]= elemento;
for(i=0;;i++)
   if(arreglo[i] == elemento)
      break;
```

Si al acabar el bucle, i es igual a $largo$, entonces el elemento no se encontraba en el arreglo.

1.2.2. Búsqueda binaria (o dicotómica)

Esta búsqueda consiste, por un lado, en dividir el arreglo a la mitad, convirtiéndolo en dos subarreglos más pequeños, y por el otro, comparar el elemento a buscar con el elemento del centro. Si ambos coinciden, la búsqueda se termina. Si el elemento es menor, el mismo estará en el primer subarreglo, pero si es mayor, en el segundo. Para ambos casos, se continúa recursivamente.

También puede darse el caso en que el elemento no se encuentre en el arreglo. Para utilizar este método, el arreglo debe estar ordenado.

Ejemplo:

Supongamos que se quiere buscar el elemento *4* en el arreglo: [1,2,3,4,5,6,7,8,9,10,11]. Entonces, se realizarían los siguientes pasos:

- Se toma el elemento central y se divide el arreglo en dos:
  [1,2,3,4,5]-6-[7,8,9,10,11]
- Como el elemento buscado *4* es menor que el central *6*, entonces debe estar en el primer subarreglo:
  [1,2,3,4,5]
- Se vuelve a dividir el arreglo en dos:
  [1,2]-3-[4,5]
- Como el elemento buscado es mayor que el central, debe estar en el segundo subarreglo:
  [4,5]
- Se vuelve a dividir en dos:
  []-4-[5]

Como el elemento buscado coincide con el central, lo hemos encontrado.

Si al final de la búsqueda todavía no lo hemos hallado, y el subarreglo a dividir está vacío [], el elemento no se encuentra en el arreglo.

La implementación sería:

```
// desde y hasta indican los límites del arreglo que se está iterando.
int desde,hasta,medio,elemento,posicion;
const int largo = 9;
int arreglo[largo] = {1,2,3,4,5,6,7,8,9};
elemento = 3;

for(desde=0,hasta=largo-1;desde<=hasta;){
        if(desde==hasta){ // si el arreglo sólo tiene un elemento:
                if(arreglo[desde]==elemento)
                        posicion=desde;
                else
                        posicion=-1; // no está en el arreglo.
                break; // Salir del bucle.
        }
        medio = (desde+hasta)/ 2;
        if(arreglo[medio]==elemento){
                posicion=medio;
                printf("Solucion : elemento = %i <-> arreglo[%i] = %i",
                        elemento,posicion, arreglo[posicion]);
                break; // y sale del bucle.
        }else if(arreglo[medio]>elemento){
                hasta=medio-1;
                printf("Elige subarreglo izquierdo \n");
        }else{
                desde=medio+1;
                printf("Elige subarreglo derecho \n");
        }
}
```

1.2.3. Búsqueda mediante claves (Hashing)

Este método es el más veloz de los analizados anteriormente, y además, no requiere que los elementos estén ordenados.

Este método consiste en asignar a cada elemento un índice mediante una fórmula de conversión, llamada *función hash.* La función más sencilla es la identidad, esto es: al número 0 se le asigna el índice 0; al elemento 1, el índice 1, y así de manera sucesiva.

Sin embargo, cuando los elementos son muy grandes, se desperdicia mucho espacio, ya que se necesitan arreglos grandes para almacenarlos.

Supongamos que tenemos los elementos en las posiciones siguientes: arreglo[0] = 0, arreglo[1] = 1, arreglo[250], arreglo[351], arreglo[10000] = 10000.

Desde las posiciones 250 a 351, o peor aún, desde 351 a 10000 del arreglo, puede no haber nunca un elemento. Entonces, se reservan espacios de memoria que tal vez no se utilicen y éstos quedan con muchos espacios libres sin usar.

Para optimizar el espacio se emplean funciones hash más complejas.

La función ideal debería ser *biyectiva,* es decir, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento, pero no siempre es posible encontrar esa función.

La función elegida depende de cada problema y las más utilizadas son:

- Truncamiento
- Plegamiento
- Aritmética modular
- Mitad del cuadrado

Truncamiento

Ignora parte de la clave y utiliza la parte restante directamente como índice.

Si, por ejemplo, las claves son enteros de 10 dígitos y la tabla de transformación tiene 10000 posiciones, entonces el segundo, cuarto, quinto y octavo dígitos podrían formar la función de conversión.

Clave: 3869574863

Índice: 8958

El truncamiento es un método rápido, pero falla para distribuir las claves de modo uniforme, lo que implica muchos huecos libres.

Plegamiento

Consiste en fragmentar la clave en diferentes partes y combinar las mismas, generalmente utilizando la suma o multiplicación, para obtener el índice. Cada parte de la clave debe contener la misma cantidad de dígitos que los índices necesarios. Es decir, si sólo podemos asignar hasta 999 índices, entonces las partes serán de tres dígitos. De esta manera, se obtiene otro número de tres dígitos; si no, se toman los últimos tres:

Clave: 87509245 => Función: 875 + 092 + 45 = 1012 => Índice: 012

Aritmética modular

En esta función, el índice es igual a: el resto de la división de la clave y un número N. Este número N debe ser preferentemente primo y con la misma cantidad de dígitos que el índice máximo establecido. Los elementos se guardarán en los índices de 0 a N-1, ya

que nunca puede darse el caso de que el resto sea mayor a N. Si N es igual al número primo 101, entonces:

Clave: 23456790 => Función: 23456790 MOD 101 = 045 => Índice: 045

Mitad del cuadrado

Consiste en elevar al cuadrado la clave y tomar las cifras centrales como índice:

Clave: 2314 => Función: 2314* 2314= 5354596 => Índice: 545

Colisiones

Las funciones hash no siempre ofrecen valores distintos, sino que es posible obtener el mismo índice para dos claves diferentes. Esta situación se denomina *colisión* y se deben encontrar métodos para su correcta resolución. Todos los métodos anteriormente analizados producen colisiones.

Ejemplo:

Si aplicamos la función de truncamiento al segundo, cuarto, quinto y octavo dígitos en las claves 3869574863 y 1859537821, obtenemos para ambas el índice 8958.

Para corregir este problema, existen varias técnicas como *Rehashing* o *Intercalación Cuadrática.* Pero lo más efectivo es que, en lugar de crear un arreglo de números, se pueda generar un arreglo de punteros, donde cada puntero señale el principio de una lista enlazada.

Entonces, cuando un elemento llega a un determinado índice, el elemento se coloca en el último lugar de la lista de ese mismo índice. El tiempo de búsqueda se reduce de manera considerable y se pueden añadir nodos dinámicamente a la lista, con lo que evita poner restricciones al tamaño del arreglo.

En los capítulos siguientes veremos este tema de las listas enlazadas.

Ejemplo de aplicación de búsqueda

Continuando con el ejemplo de ordenamiento, podríamos suponer que necesitamos encontrar una pieza particular del rompecabezas, aquella que coincida con un número de posición.

Para ello, podríamos recurrir a una función ***find_if,*** perteneciente a la biblioteca estándar de C++ (análogamente a lo realizado con sort).

Por una cuestión de simplicidad, mostramos las líneas de código añadidas:

```
//Declaración de un puntero de tipo Pieza
Pieza *p;

//Búsqueda del objeto
p = find_if(rompecabezas,rompecabezas+9,es_igual_a(125));

//Controlo que el puntero al objeto devuelto
//este dentro de un rango válido
if (p == rompecabezas + 9) {
        cout << "No se encontró el elemento buscado";
} else {
        cout << "La imagen es: " << p->getFooImg();
}
```

Cabe aclarar que la flexibilidad de `find_if` se debe a la posibilidad de aceptar un predicado como parámetro. Aquí presentamos el código del predicado `es_igual_a()`; siempre debe retornar un valor booleano:

```
class es_igual_a
{
public:
```

```
    es_igual_a (int n)
      : value(n)
    {}

    bool operator() (Pieza element) const
    {
       return element.getInitPos() == value;
    }

private:
    int value;
};
```

Este es un caso especial de predicado, pues lo declaramos como una clase, lo cual nos permite parametrizarlo, agregando el valor buscado.

Si bien el código puede parecer complejo para esta instancia, sirve para mostrar la potencia de las funciones ofrecidas por la biblioteca de C++.

1.3. Actualización de arreglos ordenados

Cuando se utilizan arreglos como parte de un algún programa, normalmente es necesario actualizar sus valores como en cualquier otra estructura de datos.

Si estos arreglos se encuentran ordenados, entonces debemos procurar que se mantengan así una vez actualizados. Por ello, tenemos que sopesar la conveniencia entre el costo de mantener un arreglo ordenado a fin de realizar una búsqueda más rápida, o por el contrario, actualizar rápidamente y acceder a una búsqueda más lenta.

Los tipos de actualización considerados incluyen inserción, modificación y eliminación. En cualquiera de los casos, es necesario encontrar la posición donde actualizar. En el primer y tercer caso se requiere, además, un corrimiento de valores para que ocupen los lugares correspondientes.

1.3.1. Inserción

La inserción en arreglos ordenados consiste en agregar un valor entre otros existentes. Como mencionamos anteriormente, primero es necesario encontrar el lugar apropiado. Según el caso, el valor a insertar puede existir o no, pero de cualquier forma es conveniente aplicar un algoritmo que aproveche el ordenamiento de los elementos. Aquí invocaremos a las funciones `find_if`, que devuelven la posición donde se debe insertar el nuevo elemento.

Como segundo paso, deberemos crear un espacio para alojar el nuevo elemento, lo cual nos obligará a correr un lugar a todos los elementos posteriores.

```
void insertar(int arr[], int n, int &sup, int val)
{
        if (sup<n-1){

                //Usamos la función find_if junto con
                //los objetos función bind2nd y greater_equal
                //de la STL
                int *p=find_if(arr,arr + n,bind2nd(greater_equal<int>(),val));
                int pos=p-arr;

                //Si el puntero devuelto va mas allá
                //de los limites, no se encontro un
                //elemento que cumpla con el predicado
                //ubicamos el elemento el la última libre
                if (p==arr+n) pos=sup+1;

                //Hacemos lugar para el nuevo elemento
                for(int i=sup+1; i>=pos;i--)
                        arr[i]=arr[i-1];
                arr[pos]=val;
                sup++;
        }
}
```

1.3.2. Modificación

La modificación de un valor presume la existencia previa de un valor. Por lo tanto, al igual que el caso anterior, es necesario buscar el elemento, aunque tenemos la ventaja de que no hay que hacer corrimiento de lugares.

```
void modificar(int arr[], int n, int oldval, int newval)
{
        //Usamos la función find
        int *p =find(arr,arr + n,oldval);
        int pos =p-arr;
        if (p!=arr+n) arr[pos]=newval;
}
```

1.3.3. Eliminación

La eliminación es similar a la inserción en cuanto a que debemos hacer una búsqueda y posterior reubicación, pero en este caso es necesario quitar un lugar en el arreglo.

```
void eliminar(int arr[], int n, int sup, int val)
{
        //Usamos la función find
        int *p=find(arr,arr + n,val);
        int pos=p-arr;

        if (p!=arr+n){
                for(int i=pos; i<=sup;i++)
                        arr[i]=arr[i+1];
                arr[sup]=NULL;
        }
}
```

Ejemplo de actualización de arreglos

Estas operaciones son ampliamente aplicables en la mayoría de nuestros juegos. Imagínense, por ejemplo, el clásico esquema en el que poseemos una nave que tiene un arma con la que podemos derribar a los enemigos.

Para programar esta lógica, debemos usar varias de las acciones que hemos visto.

Llegado el momento de agregar un enemigo, ya sea por el paso del tiempo o nivel, podríamos tener una secuencia (muy simplificada) como la que sigue:

```
Enemigo crearEnemigo(int x, int y)
{
        Enemigo nuevoEnemigo;
        nuevoEnemigo.X=x;
        nuevoEnemigo.Y=y;
        insertarEnemigo(enemigos,6,nuevoEnemigo);
        return Enemigo;
}
```

Por el contrario, si hemos aniquilado a una de las naves invasoras, debemos sacarla de la memoria:

```
int derribarEnemigo(Enemigo enemigoDerribado)
{
        eliminarEnemigo(enemigos,50, enemigoDerribado);
        return 0;
}
```

BIBLIOGRAFÍA

- Schildt, H., McGraw-Hill. *C++: A Beginner's Guide,* Second Edition, ISBN-13: 978-0072232158.

- Sedgewick,R., *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching,* Third Edition , Addison Wesley Professional, ISBN-13: 978-0-201-35088-3.

- C++ con Clase, http://c.conclase.net/

- cplusplus.com, http://www.cplusplus.com/

- Algoritmia.net, http://www.algoritmia.net/articles.php?id=32