Arreglos y ordenación

Ing José Luis MARTÍNEZ

30 de agosto de 2024

Arrays y ordenación

1. Arrays

Un array (arreglo) en C++, al igual que en C, almacena muchos elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. El array es muy importante por diversas razones. Una de ellas es almacenar secuencias o cadenas de texto. Utilizando el tipo array, se puede crear una variable que contenga un grupo de caracteres

Un array (arreglo, lista o tabla) es una secuencia de objetos del mismo tipo, que se numeran consecutivamente 0, 1, 2, 3. Estos números se denominan valores índice o subíndice del array. Cada ítem del array se denomina elemento. El tipo de elementos del array puede ser cualquier tipo de dato de C++, incluyendo estructuras definidas por el usuario. Si el nombre del array es a, entonces a[0] es el nombre del elemento que está en la posición 0, a[1] es el nombre del elemento que está en la posición 1, etc. En general, el elemento i-ésimo está en la posición i-1. De modo que si el array tiene n elementos, sus nombres son a[0], a[1],...,a[n-1] . Un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el tamaño o longitud del array. Para indicar el tamaño o longitud del array se debe hacer se guir al nombre, el tamaño encerrado entre corchetes. La sintaxis para declarar un array de una dimensión determinada es:
tipo nombreArray[numeroDeElementos];

El índice de un array se denomina, con frecuencia, subíndice del array. El método de numeración del elemento i-ésimo con el índice o subíndice i - 1 se denomina indexación basada en cero. Todos los subíndices de los arrays comienzan con 0. Su uso tiene el efecto de que el índice de un elemento del array es siempre el mismo que el número de "pasos"desde el elemento inicial a[0] a ese elemento. Por ejemplo, a[4] está a 4 pasos o posiciones del elemento a[0] . En los programas se pueden referenciar elementos del array utilizando fórmulas o expresiones enteras para los subíndices.

Los elementos de los arrays se almacenan en bloques contiguos de memoria. Los arrays que tienen un solo subíndice se conocen como arrays unidimensionales (una sola dimensión). Cada bloque de memoria tiene el tamaño del tipo de dato. Hay que tener en cuenta que C++ no comprueba que los índices del array están dentro del rango definido, por lo que debe ser controlado por el usuario. Se debe tener cuidado de no asignar valores fuera del rango de los subíndices, debido a que se sobreescribirían datos o código.

Ejemplo. Posiciones válidas de un array. En la declaración del array int a[7] los índices válidos son a[0], a[1],.., a[6]. Pero si se pone a[10] no se da mensaje de error y el resultado puede ser impredecible.

Ejemplo. Posiciones ocupadas por los elementos de un array. Si a es un array de números reales y cada número real ocupa 4 bytes, entonces si el elemento a[0] ocupa la dirección del elemento a[i], ocupa la dirección de memoria d + (i – 1) * 4. El operador *sizeof* devuelve el número de bytes necesarios para contener su argumento. Al usar *sizeof* para solicitar el tamaño de un array, se obtiene el número de bytes reservados para el array completo. Conociendo el tipo de dato almacenado en el array y su tamaño, se puede calcular la longitud del array , mediante el cociente *sizeof(a)*/tamaño(dato) .

Ejemplo. Protección frente a errores en el intervalo (rango) de valores de una variable de índice que representa un array. Si n está fuera de rango se retorna error, en otro caso se calcula la media de los números almacenados en el array.

```
float media ( double m[], int n)
{
   if (n * sizeof(float) > sizeof(m))
      return - 32767; // error
   float Suma = 0;
   for( int i = 0; i < n; i++)
      Suma += m[i];
   return Suma / n;
}
```

Ejemplo. Almacenamiento en memoria de un array de 5 elementos. Se declara el array a de 5 elementos y se almacenan en las posiciones de memoria los 5 primeros números pares positivos.

```
int a[4];
for( int i = 0; i < 5; i++)
   a[i] = 2 * i + 2;
```

Posición índice	0	1	2	3	4
Valor almacenado	2	4	6	8	10

Al declarar los arreglos pueden presentarse las siguiente opciones:

a. Se declara el arreglo con su tamaño, por ej. *int a[7]*, es un arreglo de enteros de 7 elementos

b. Se declara el arreglo con valores iniciales, por eje *float b[3]={ 1.0, 3.4, 5.6 };*

c. Se declara el arreglo sin indicar su tamaño pero dándole valores iniciales, por ej. *char* $nombre[] = \{"Juan"\}$*;*

Ejemplo. Asignación de cadenas de caracteres. Las cadenas de caracteres pueden asignar se en la inicialización de un array o mediante la función de la biblioteca estándar strcpy() (çopiar cadenas") permite copiar una constante de cadena en una cadena. Sin embargo, no puede asignar se un valor a una cadena con la asignación =.

```
char Cadena[6];
strcpy(Cadena,"abcde");
```

Es equivalente a la siguiente declaración.

```
char Cadena[6] = "abcde";
```

Las cadenas finalizan con \0, indica caracter nulo (NULL).

0	1	2	3	4	5
a	b	c	d	e	\0

1.1. Arreglos Multidimensionales

Los arrays multidimensionales más usuales son los de dos dimensiones, conocidos también por el nombre de tablas o matrices. Sin embargo, es posible crear arrays de tantas dimensiones como requieran sus aplicaciones, ésto es, tres, cuatro o más dimensiones. Por ejemplo una fotografía digital es un arreglo de tres dimensiones, donde cada pixel tiene una posición horizontal, una posición vertical y una profundidad correspondiente a los colores R(red), G(green), B(blue).
La sintaxis para la declaración de un array de dos dimensiones es:

<TipoElemento><nombrearray>[<NúmeroDeFilas<][<NúmeroDeColumnas>]

Un array de dos dimensiones en realidad es un array de arrays. Es decir, es un array unidimensional, y cada elemento no es un valor entero de coma flotante o carácter, sino que cada elemento es otro array. La sintaxis para la declaración de un array de tres dimensiones es:

<tipodedatoElemento><nombrearray>[<Cota1>] [<Cota2>][<Cota3>

Ejemplo. Declaración y almacenamiento de un array bidimensional en memoria. La declaración de un array de enteros que tiene 2 filas y 3 columnas, su representación y almacenamiento por filas es el siguiente:

int a[2][3];

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

Por ejemplo inicialización de arrays bidimensionales.

```
int ejemplo[2][3] = {1,2,3,4,5,6};
int ejemplo [2][3] = { {1,2,3}, {4,5,6}}
```

Los arrays multidimensionales (si no son globales) no se inicializan a valores específicos a menos que se asignen valores en el momento de la declaración o en el programa. Si se inicializan uno o más elementos, pero no todos, C++ rellena el resto con ceros o valores nulos (\0). Si se desea inicializar a cero un array multidimensional, utilice una sentencia tal como ésta: *float a[3][4] = 0.0;*

El formato general para asignación directa de valores a los elementos de un array bidimensional es la siguiente:

<nombre array>[indice fila][indice columna] = valor elemento;

Para la extracción de elementos:

<variable>= <nombre array>[indice fila][indice columna];

Las funciones de entrada o salida se aplican de igual forma a los elementos de un array unidimensional. Se puede accedera los elementos de arrays bidimensionales mediante bucles anidados. Su sintaxis general es:

```
int IndiceFila, IndiceCol;
for (IndiceFila = 0; IndiceFila < NumFilas; ++IndiceFila)
   for (IndiceCol = 0; IndiceCol < NumCol; ++IndiceCol)
      Procesar-elemento[IndiceFila][IndiceCol];
```

Ejemplo. Lectura y visualización de un array de 3 filas y 5 columnas. Se declaran las constantes *maxf a 3* y *maxc a 5* y, posteriormente, se declara el array bidimensional a y se lee y visualiza por filas.

```
#include <cstdlib>
#include <iostream>
#define maxf 3
#define maxc 5
using namespace std;
int main( int argc, char *argv[])
{
   float a[maxf] [maxc];
   int f, c;
   // leer el array
   for(f = 0; f < maxf; f++)
      for(c = 0; c < maxc; c++)
         cin >> a[f][c];
   // escribir el array
   for(f = 0; f < maxf; f++)
   {
      for(c = 0; c < maxc; c++)
         cout << a[f] [c] ;
      cout << endl;
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Ejemplo. Juego guerra naval.

```
/*Ejemplo de arreglo en 2D - Guerra naval*/
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
   char flota[10][10]={{},{}};// arreglo de 10 x 10 char
   for(int i=0; i < 10; i++)
   {
      for(int j=0; j < 10; j++)
      {
         flota[i][j] = '~';
      }
   }
   cout << "Escenario antes de la batalla. ") << endl;
   for(int i=0; i < 10; i++)
   {
      for(int j=0; j < 10; j++)
      {
         cout << flota[i][j];
      }
      cout << endl;
   }
   /* ----------------------------------------------------------
   -- Ejemplo de ubicación de un elemento
   ----------------------------------------------------------*/
   cout << "Escenario al colocar unos barcos." << endl;
   flota[1][1]='B';
   flota[3][2]='B';
   flota[4][6]='B';
   for(int i=0; i < 10; i++)
   {
      for(int j=0; j < 10; j++)
      {
         cout << flota[i][j];
      }
      cout << endl;
   }
   return 0;
}
```

Ejemplo Array tridimensional. Se declara un array tridimensional para almacenar las temperaturas de cada uno de los 60 minutos de las 24 horas de un mes de 31 días. Se declaran como constantes dias, horas y minutos.
En tres bucles anidados se leen la temperatura de cada uno de los 60 minutos, de las 24 horas, de los 31 días de un mes. Posteriormente, se calcula y se escribe la media de temperatura de cada día.

```
#include <cstdlib>
#include <iostream>
#define dias 31
#define horas 24
#define minutos 60
```

```
using namespace std;
int main( int argc, char *argv[])
{
    int i, j, k;
    float A[dias][horas][minutos], media;
    //lectura de las temperaturas.
    for(i = 0; i < dias; i++)
        for (j = 0; j < horas;j++)
            for (k = 0; k < minutos; k++)
                {
                    cout << "Ingrese la temperatura del dia " << i << " hora " << j
                    << " minuto " << k << " : ";
                    cin >> A[i][j][k];
                    cout << endl;
                }


    for( i = 0; i < dias; i++)
        { //cálculo de la media de cada uno de los días.
            media = 0;
            for ( j = 0 ; j < horas; j++)
                for ( k = 0; k < minutos; k++)
                media += A[i][j][k];
            cout<<" dia " << i+1 << " temperatura media " << media /(horas * minutos)
            << endl;
        }
    cin.get();
    return EXIT_SUCCESS;
}
```

1.2. Utilización de arrays como parámetros

En C++ todos los arrays se pasan por referencia (dirección). C++ trata automáticamente la llamada a la función como si hubiera situado el operador de dirección & delante del nombre del array. La declaración en la función de que un parámetro es un array se realiza:

<tipo de datoElemento><nombre array>[<Cota1>]. O mejor
<tipo de datoElemento><nombre array>[]. O bien
<tipo de datoElemento><nombre array>*

En este segundo caso es conveniente declarar otro parámetro que indique el tamaño del array. Tenemos por ejemplo

float suma(float a[5]);
float calcula(float a[], int n); //n es número de datos no obligatorio
*float media (float * a, int n); //n es no obligatorio, pero conveniente*

Dadas la declaraciones de arrays siguientes:

int b[5], a[6];

Son posibles llamadas válidas:

```
cout << suma(a);
cout << calcula (b, 5);
cout << media (b, 5);
```

Ejemplo. Lectura escritura de un vector con funciones. El número de elementos del vector es indicado a cada función en el parámetro n, y el vector se declara en el parámetro con corchetes vacíos. El programa declara una constante max para dimensionar los vectores. Se codifican las funciones lee, escribe, que leen y escriben respectivamente un vector de n datos que reciben como parámetros, así como la función suma que recibe como parámetros dos vectores de n datos, y calcula el vector suma almacenado el resultado en c. El programa principal llama a las funciones anteriores.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#define max 11
using namespace std;
void lee( float a[], int n);
void escribe ( float a[], int n);
void suma( float a[], float b[], float c[], int n);
int main( int argc, char *argv[])
{
   int n;
   float a[max], b[max],c[max];
   n = 3;
   lee(a, n);
   lee(b, n);
   cout << " vector a\n";
   escribe(a, n);
   cout << " vector b\n";
   escribe(b, n);
   suma (a, b, c, n);
   cout << " vector suma \n";
   escribe(c, n);
   system("PAUSE");
   return EXIT_SUCCESS;
}

void suma( float a[], float b[], float c[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      c[i] = a[i] + b[i];
}


void escribe( float a[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      cout << a[i] << " " ;
   cout << endl;
}

void leer( float a[], int n)
{
   int i;
   for (i = 0; i < n; i++)
   {
```

```
      cout << " dame dato posicion i =" << i + 1 << " ";
      cin >> a[i];
   }
}
```

1.3. Paso de matrices como parámetros

Para arrays bidimensionales es obligatorio indicar el número de columnas en la declaración del numero de parámetros de la siguiente forma:

<tipo de datoElemento><nombre array>[<Cota1>][cota2]. O bien,
<tipo de datoElemento><nombre array>[][cota2]

En este segundo caso es también con veniente declarar otro parámetro que indique el número de filas. La llamada a una función que tenga como parámetro un array se realiza con el nombre del array .

void matriz (float M[3][5]);
void matriz1(float M[][5], int nf);//nf no obligatorio, pero conveniente

Ejemplo. Función que realiza la suma de dos matrices

```
#define max 11

void Suma( float a[][max] , float b[][max] , float c[][max], int n)
{
   for ( int i = 0; i < n; i++)
      for ( int j = 0; j < n; j++)
         c[i][j] = a[i][j] + b[i][j];
}
```

1.4. Paso de cadenas como parámetros

La técnica de pasar arrays como parámetros se utiliza para pasar cadenas de caracteres a funciones. Las cadenas terminadas en nulo (\0) utilizan el primer método dado anteriormente para controlar el tamaño de un array .

Ejemplo. Función que recibe como parámetro un número entero como cadena de caracteres y convierte la cadena de caracteres en un número entero. La variable cadena sirve para leer el número introducido por teclado en una variable tipo cadena de caracteres de longitud máxima 80. La función valor_numerico, convierte la cadena de caracteres en un número entero con su signo. Esta función se salta todos los blancos y tabuladores que se introduzcan antes del número entero que es un dato leído previamente. Para convertir la cadena en número se usa la descomposición: $'2345' = 2*1000+3*100+4*10+5$. Estas operaciones se realizan por el método de Horner de evaluación de polinomios: $2345 = (((0*10+2)*10+3)*10+4)*10+5$. La obtención, por ejemplo del número 3 se hace a partir del carácter $'3' : 3 = '3' - '0'$.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int valor_numerico( char cadena[]);
int main( int argc, char *argv[])
{
   char cadena[80];
   cout << "dame numero: ";
   cin >> cadena;
   cout << " numero como cadena " << cadena << endl;
   cout<<" valor leido como numero \n "<<valor_numerico(cadena)<< endl;
   system("PAUSE");
```

```
   return EXIT_SUCCESS;
}

int valor_numerico( char cadena[])
{
   int i, valor, signo;
   // salto de blancos y tabularores
   for (i = 0; cadena[i] ==' ' || cadena[i] == '\t'; i++);
   //determinación del signo
      signo = 1;
   if(cadena[i] == '+' || cadena[i] == '-')
      signo = cadena[i++] == '+' ? 1:-1;
   // conversión a número
   for ( valor = 0 ; cadena[i] >= '0' && cadena[i] <= '9' ; i++)
      valor = 10 * valor + cadena[i] - '0';
   return (signo * valor);
}
```

2. Problemas

1. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
   int i, Primero[21];
   for i = 1; i <= 6; i++)
      cin >> Primero[i];
   for(i = 3; i > 0; i--)
      cout << Primero[2 * i];
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

2. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
   int i,j,k, Primero[10];
   for( i = 0; i < 10; i++)
      Primero[i] = i + 3;
   cin >> j >> k;
   for(i = j; i <= k; i++)
      cout << Primero[i] << " ";
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

3. Un vector se dice que es simétrico si el elemento que ocupa la posición i_ésima coincide con el que ocupa la posición $n - i_{ésima}$, siempre que el número de elementos que almacene el vector sea n. Por ejemplo el vector que almacena los valores 2, 4, 5, 4, 2 es simétrico. Escribir una función que decida si el vector de n datos que recibe como parámetro es simétrico.

4. Escribir un algoritmo que calcule y escriba una tabla con los 100 primeros números primos. Un número es primo si sólo tiene por divisores la unidad y el propio número.

5. Escribir una función que encuentre el elemento mayor y menor de una matriz, así como las posiciones que ocupa y los visualice en pantalla.

6. Codificar un programa C++ que permita visualizar el triángulo de Pascal:

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5   10  10  5   1
```

   En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando primeramente un array bidimensional y, posteriormente, un array de una sola dimensión.

7. Modifique el programa de guerra naval de tal forma que:

   a) Pueda agregar los barcos en forma aleatoria, según la cantidad tradicional en el juego, marcando su posición con una B.

   b) Pueda agregar los barcos en forma manual.

   c) Ingresar las coordenadas y que el juego le diga si impactó en un barco o hizo agua, en este último caso marcar la posición con una x.

   d) Modificar el escenario original que contiene solo agua, con los impactos errados (x) y los impactos en barcos (B)

3. Búsqueda y ordenación

3.1. Búsqueda

El proceso de encontrar un elemento específico de un array se denomina búsqueda. Este proceso permite determinar si un array contiene un valor que coincida con un cierto valor clave. La búsqueda permite la recuperación de datos previamente almacenados. Si el almacenamiento se realiza en memoria, la búsqueda se denomina interna. Las búsquedas más usadas en listas son:
secuencial (lineal) y binaria.

3.1.1. Búsqueda secuencial

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado clave. En una búsqueda secuencial (a veces llamada búsqueda lineal), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro (de izquierda a derecha o viceversa). La búsqueda lineal consiste en recorrer cada uno de los elementos hasta alcanzar el final de la lista de datos. Si en algún lugar de la lista se encontrara el elemento buscado, el algoritmo deberá informar sobre la o las posiciones donde ha sido localizado.

Ejemplo. Búsqueda secuencial de una clave en un vector de n datos de derecha a izquierda comenzando en la última posición.

```
int busquedaLinealDI ( int V [], int n, int clave)
{
   int i;
   for (i = n - 1; i >= 0; i--)
   {
      if (V[i] == clave)
         return i;
   }
   return -1;
}
```

3.1.2. Búsqueda binaria

Este método de búsqueda requiere que los elementos se encuentren almacenados en una estructura de acceso aleatorio de forma ordenada, es decir clasificados, con arreglo al valor de un determinado campo. En general, si la lista está ordenada se puede acortar el tiempo de búsqueda. La búsqueda binaria consiste en comparar el elemento buscado con el que ocupa en la lista la posición central y, según sea igual, mayor o menor que el central, parar la búsqueda con éxito, o bien, repetir la operación considerando una sublista formada por los elementos situados entre el que ocupa la posición central+1 y el último, ambos inclusive, o por los que se encuentran entre el primero y el colocado en central–1, también ambos inclusive. El proceso termina con búsqueda en fracaso cuando la sublista de búsqueda se quede sin elementos.

Ejemplo. Búsqueda binaria de una clave en un vector de n datos ordenado crecientemente.

```
int BusquedaBinaria(float V[], int n, float clave)
{
   int izquierda = 0, derecha = n - 1, central;
   bool encontrado = false;
   while((izquierda <= derecha) && (!encontrado))
   {
      central = (izquierda + derecha) / 2;
      if (V[central] == clave)
         encontrado = true; // éxito en la búsqueda
      else if (clave < V[central]) //a la izquierda de central
         derecha = central - 1 ;
      else // a la derecha de central
         izquierda = central + 1;
   }
   return encontrado ? central:-1; //central si encontrado -1 otro caso
}
```

3.2. Ordenación

La ordenación o clasificación de datos es la operación consistente en disponer un conjunto de datos en algún orden determinado con respecto a uno de los campos de elementos del conjunto. La clasificación interna es la ordenación de datos que se encuentran en la memoria principal del ordenador. Una lista dice que está ordenada por la clave k si la lista está en orden ascendente o descendente con respecto a esta clave. La lista se dice que está en orden ascendente si:

i <j implica que lista[i] <= lista[j]

y se dice que está en orden descendente si:

i >j implica que lista[i] <= lista[j].

Los métodos de clasificación interna se dividen en dos tipos fundamentales al evaluar su complejidad en cuanto al tiempo de ejecución:

- Directos. De complejidad $O(n^2)$ como burbuja (intercambio directo), selección e inserción.
- Avanzados. De complejidad inferior $O(n2)$ como Shell, y otros de complejidad $O(nlogn)$, como el método del montículo, ordenación por mezcla o radix Sort.

Estudiaremos los métodos directos y el método Shell.

3.2.1. Ordenación por burbuja

La ordenación por burbuja ("buble sort") se basa en comparar elementos adyacentes de la lista (vector) e intercambiar sus valores si están desordenados. De este modo, se dice que los valores más grandes burbujean hacia la parte superior de la lista (hacia el último elemento), mientras que los valores más pequeños se hunden hacia el fondo de la lista. En el caso de un array (lista) con n elementos, la ordenación por burbuja requiere hasta n–1 pasadas.

La primera pasada realiza los siguientes pasos: el vector tiene los elementos A[0], A[1], ..., A[n – 1] . El método comienza comparando A[0] con A[1] ; si están desordenados, se intercambian entre sí. A continuación, se compara A[1] con A[2] , realizando el intercambio entre ellos si están desordenados. Se continua comparando A[2] con A[3] , intercambiándolos si están desordenados..., hasta comparar A[n – 2] con A[n – 1] intercambiándolos si están desordenados. Al terminar esta pasada el elemento mayor está en la parte superior de la lista.

La segunda pasada realiza un proceso parecido al anterior , pero la última comparación y, por tanto, el último posible intercambio es realizado con los elementos A[n – 3] y A[n – 2] y, por tanto, coloca el segundo elemento mayor en la posición A[n – 2]. El proceso descrito se repite durante n – 1 pasadas teniendo en cuenta que en la pasada i se ha colocado el elemento mayor de las posiciones 0, ..., n – i en la posición n – i . De esta forma, cuando i toma el valor n – 2, el vector está ordenado.

Ejemplo. Codificación del método de la burbuja. Se realiza la ordenación poniendo en primer lugar el elemento mayor en la última posición del array . Posteriormente, se coloca el siguiente mayor en la penúltima posición, y así sucesivamente. Para realizar el proceso sólo se utilizan comparaciones de elementos consecutivos, intercambiándolos en el caso de que no estén colocados en orden.

```
void Burbuja( float A[], int n)
{
int i, j;
float auxiliar;
for (i = 0; i < n - 1; i++) // n-1 pasadas
   for (j = 0; j < n - 1 - i; j++) // burbujeo de datos
      if (A[j] > A[j + 1]) //comparación de elementos contiguos
      { // mal colocados intercambio
         auxiliar = A[j];
         A[j] = A[j + 1];
         A[j + 1] = auxiliar;
      }
}
```

En este método de ordenación, los dos bucles comienzan en cero y terminan en n – 2 y n – i – 1. El método de ordenación requiere como máximo n – 1 pasadas a través de la lista o el array. Durante la pasada 1, se hacen n – 1 comparaciones y como máximo n – 1 intercambios, durante la pasada 2, se hacen n – 2 comparaciones y como máximo n – 2 intercambios. En general, durante la pasada i , se hacen n – i comparaciones y a lo más n – i intercambios. Por consiguiente, en el peor caso, habrá un total de $(n–1) + (n–2) + ... + 1 = n * (n–1)/2 = O(n^2)$ comparaciones y el mismo número de intercambios. Por consiguiente, la eficiencia del algoritmo de burbuja en el peor de los casos es $O(n^2)$.

Ejemplo. Seguimiento del método de ordenación de burbuja para los datos 8, 3, 8, 7, 14, 6. Si ejecutamos y mostramos paso a paso el método obtendremos

```
#include <iostream>

using namespace std;
void mostrarVector(int v[], int n);
void ordenBurbuja(int v[], int n);
int main()
{
    int vector[6]={8, 3, 8, 7, 14, 6};
    cout << "El vector desordenado es. " << endl;
    mostrarVector(vector, 6);
    ordenBurbuja(vector, 6);
    return 0;
}

void mostrarVector(int v[], int n)
{
    for(int i=0; i<n; i++)
        cout << v[i] << ' ';
    cout << endl;
}

void ordenBurbuja(int v[], int n)
{
int i, j;
int auxiliar;
for (i = 0; i < n-1; i++) // n-1 pasadas
{
   for (j = 0; j < n-1-i; j++) // burbujeo de datos
   {
      if (v[j] > v[j + 1]) //comparación de elementos contiguos
      { // mal colocados intercambio
         auxiliar = v[j];
         v[j] = v[j + 1];
         v[j + 1] = auxiliar;
      }
   }
   mostrarVector(v, n);
}
}
```

3.2.2. Ordenación por selección

El método de ordenación se basa en seleccionar la posición del elemento más pequeño del array y su colocación en la posición que le corresponde. El algoritmo de selección se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista. El algoritmo de ordenación por selección de una lista (vector) de n elementos se realiza de la siguiente forma: se encuentra el elemento menor de la lista y se intercambia el elemento menor con el elemento de subíndice 0. A continuación, se busca el elemento menor en la sublista de subíndices 1 .. n – 1, y se intercambia con el elemento de subíndice 1. Después, se busca el elemento menor en la sublista 2 .. n – 1 y se intercambia con la posición 2. El proceso continúa sucesivamente, durante n–1 pasadas. Una vez terminadas las pasadas la

lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

Ejemplo. Codificación del método de ordenación por selección. Como primera etapa en el análisis del algoritmo se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. La función siempre realiza n – 1 intercambios (n, número de elementos del array), ya que hay n – 1 llamadas a intercambio). El bucle interno hace n – 1 – i comparaciones cada vez; el bucle externo va desde 0 a n – 2 de modo que el número total de comparaciones C es $O(n^2)$, por lo que el número de comparaciones de clave es cuadrático. Ha de observarse que el algoritmo no depende de la disposición inicial de los datos, lo que supone una ventaja de un algoritmo de selección ya que el número de intercambios de datos en el array es lineal. $O(n)$.

Seguimiento del método de ordenación de selección para los datos 8, 3, 8, 7, 14, 6.

```
#include <iostream>

using namespace std;
void mostrarVector(int v[], int n);
void ordenSelecc(int v[], int n);
int main()
{
    int vector[6]={8, 3, 8, 7, 14, 6};
    cout << "El vector desordenado es. " << endl;
    mostrarVector(vector, 6);
    cout << "Ordenamiento del vector." << endl;
    ordenSelecc(vector, 6);
    return 0;
}

void mostrarVector(int v[], int n)
{
    for(int i=0; i<n; i++)
        cout << v[i] << ' ';
    cout << endl;
}

void ordenSelecc( int v[], int n)
{
    int i, j, indicemin;
    int auxiliar;
    for (i = 0; i < n-1; i++)
    {
        indicemin = i; // posición del menor
        for(j = i + 1; j < n; j++)
            if(v[j] < v[indicemin])
                indicemin = j; // nueva posición del menor
        auxiliar = v[indicemin]; // intercambia posiciones
        v[indicemin] = v[i];
        v[i] = auxiliar;
        mostrarVector(v, n);
    }
}
```

Como primera etapa en el análisis del algoritmo se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. La función siempre realiza n – 1 intercambios (n, nú-

mero de elementos del array), ya que hay n – 1 llamadas a intercambio). El bucle interno hace n – 1 – i comparaciones cada vez; el bucle externo va desde 0 a n – 2 de modo que el número total de comparaciones C es $O(n^2)$, por lo que el número de comparaciones de clave es cuadrático.

Ha de observarse que el algoritmo no depende de la disposición inicial de los datos, lo que supone una ventaja de un algoritmo de selección ya que el número de intercambios de datos en el array es lineal $O(n)$.

3.2.3. Ordenación por inserción

Este método consiste en tomar una sublista inicial con un único elemento que se podrá considerar siempre ordenada. En la sublista ordenada se irán insertando sucesivamente los restantes elementos de la lista inicial, en el lugar adecuado para que dicha sublista no pierda la ordenación. El algoritmo considera que la sublista A[0], A[1], ..., A[i–1] está ordenada, posteriormente inserta el elemento A[i] en la posición que le corresponda para que la sublista A[0] , A[1] , ... A[i–1] , A[i] esté ordenada, moviendo hacia la derecha los elementos que sean necesarios. La sublista ordenada aumentará su tamaño cada vez que se inserta un elemento hasta llegar a alcanzar el de la lista original, momento en el que habrá terminado el proceso. Los métodos de ordenación por inserción pueden ser directo o binario, dependiendo de que la búsqueda se realice de forma secuencial o binaria.

Ejemplo. Codificación del método de ordenación por inserción lineal. La sublista 0 .. i – 1 está ordenada, y se coloca el elemento que ocupa la posición i de la lista en la posición que le corresponde mediante una búsqueda lineal. De esta forma, la sublista se ordena entre las posición 0 .. i . Si i varía desde 1 hasta n – 1 se ordena la lista de datos.

```
void Insercionlineal( float A[], int n)
{
   int i,j;
   bool encontrado;
   float auxiliar;
   for (i = 1; i < n; i++)
   { // A[0], A[1], ..., A[i-1] está ordenado
      auxiliar = A[i];
      j = i - 1;
      encontrado = false;
      while (( j >= 0 ) && !encontrado)
         if (A[j] > auxiliar)
         { // se mueve dato hacia la derecha para la inserción
            A[j + 1] = A[j];
            j--;
         }
         else
            encontrado = true;
         A[j + 1] = auxiliar;
   }
}
```

Ejemplo. Codificación del método de ordenación por inserción binaria. La única diferencia de los dos métodos de ordenación por inserción, radica en el método de realizar la búsqueda. En este caso es una búsqueda binaria, por lo que el desplazamiento de datos hacia la derecha debe hacerse después de haber encontrado la posición donde se debe insertar el elemento, que en esta codificación es siempre izquierda.

```
void Insercionbinaria( float A[], int n )
{
   int i, j, izquierda, derecha, centro;
   float auxiliar;
```

```
   for(i = 1; i < n; i++)
   {
      auxiliar = A[i];
      izquierda = 0;
      derecha = i - 1;
      while (izquierda <= derecha) //búsqueda binaria sin interruptor
      {
         centro = ( izquierda +derecha) / 2;
         if(A[centro] > auxiliar)
            derecha = centro - 1;
         else
            izquierda = centro + 1;
      }
      // desplazar datos hacia la derecha
      for(j = i - 1; j >= izquierda; j--)
         A[j + 1] = A[j];
      A[izquierda] = auxiliar;
   }
}
```

3.2.4. Ordenación Shell

La ordenación Shell se suele denominar también ordenación por inserción con incrementos decrecientes. Es una mejora de los métodos de inserción directa y burbuja, en el que se comparan elementos que pueden ser no contiguos. La idea general de algoritmo es la siguiente. Se d vide la lista original (n elementos) en $n/2$ grupos de dos elementos, con un intervalo entre los elementos de cada grupo de $n/2$ y se clasifica cada grupo por separado (se comparan las parejas de elementos y si no están ordenados se intercambian entre sí de posiciones). Se divide ahora la lista en $n/4$ grupos de cuatro con un intervalo o salto de $n/4$ y, de nuevo, se clasifica cada grupo por separado. Se repite el proceso hasta que, en un último paso, se clasifica el grupo de n elementos. En el último paso el método Shell coincide con el método de la burbuja.

Ejemplo. Codificación del método de ordenación Shell.

```
#include <iostream>

using namespace std;
void mostrarVector(int v[], int n);
void ordenShell(int v[], int n);
int main()
{
    int vector[6]={8, 3, 8, 7, 14, 6};
    cout << "El vector desordenado es. " << endl;
    mostrarVector(vector, 6);
    cout << "Ordenamiento del vector." << endl;
    ordenShell(vector, 6);
    return 0;
}

void mostrarVector(int v[], int n)
{
    for(int i=0; i<n; i++)
        cout << v[i] << ' ';
    cout << endl;
}
```

```
void ordenShell( int v[], int n)
{
    int i, j, k, salto = n/2;
    float auxiliar;
    while (salto > 0) // ordenación de salto listas
    {
       for (i = salto; i < n; i++) // ordenación parcial de lista i
       { // los elementos de cada lista están a distancia salto
          j = i-salto;
          while(j >= 0)
          {
             k = j + salto;
             if (v[j] <= v[k]) // elementos contiguos de la lista
                j = -1; // fin bucle par ordenado
             else
             {
                auxiliar = v[j];
                v[j] = v[k];
                v[k] = auxiliar;
                j = j - salto;
             }
          }
          mostrarVector(v, n);
       }
       salto = salto / 2;
    }
}
```

3.3. La clase <vector>de la biblioteca estándar de plantillas

Una de las características más importantes que debe tener un programa es poder ser reutilizado la mayor cantidad de veces posible. En los algoritmos de búsqueda, en la definición de arreglos, en los tipos de datos que se van a utilizar; si encargamos a varios programadores que realicen funciones y retazos de código, es bastante probable que no sean compatibles entre sí. Por eso C++ tiene incorporada una biblioteca estándar de plantillas, donde se incorporan varias funciones de utilización general, con el objetivo de que el código sea universal y reutilizable.

La clase *vector* es un arreglo unidimensional que puede cambiar su tamaño. Al igual que los arreglos los vectores almacenan los datos en direcciones contiguas de memoria, por lo tanto sus elementos se pueden acceder en forma aleatoria o mediante aritmética de punteros (tema que veremos más adelante). A diferencia de los arreglos que tienen un tamaño fijo, este puede cambiar.

Comparados con los arreglos los vectores utilizan más memoria, a cambio de poder almacenar cualquier tipo de datos y crecer dinámicamente de forma eficiente.

Para utilizar los vectores se debe agregar la biblioteca

#include <vector>

Para definir un vector

vector<T >variable;

Donde en T se debe colocar el tipo de variable de que estará hecho el vector, la gran flexibilidad está dada en que puede colocar cualquier tipo de dato estándar (int, char, float, double) o datos definidos por el usuario (empleado, equipo, resistencias, etc).

Para ingresar un dato se lo hace con *pushback()* como se muestra en el ejemplo siguiente.

```
#include <iostream>
```

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

int main()
{
    vector<int> x;
    x.push_back(6);
    x.push_back(5);
    x.push_back(23);
    x.push_back(12);
//    sort(x.begin(), x.end());
    for(int i=0; i < x.size(); i++)
        cout << x[i] << ", ";
    cout << endl;

    return EXIT_SUCCESS;
}
```

La salida del programa es

6, 5, 23, 12,

Se puede observar en el programa que se ha incorporado una librería de nombre *algorithm* la cual contiene varios algoritmos utilizados en programación, entre ellos está el de ordenación *sort*, si valido el renglón correspondiente a la ordenación, la salida del vector es

5, 6, 12, 23,

Con el uso iremos aprendiendo las distintas utilidades de estas bibliotecas.

Del programa podemos agregar que *x.begin()* es el primer elemento del vector *x.end()* es el último, *x.size()* devuelve el tamaño del vector. En la figura 1 se observa que al escribir el nombre del vector seguido por un punto, se despliega un menú de funciones que pueden ser utilizadas. [1] [2]

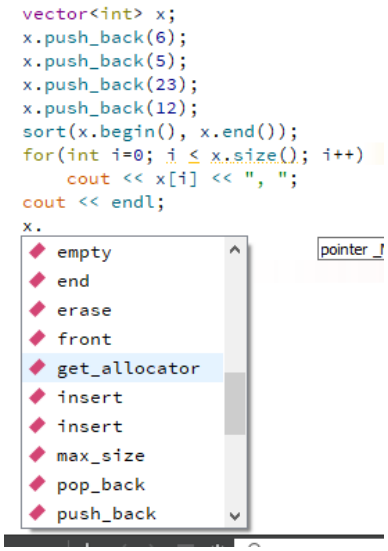

Figura 1: Opciones de función que muestra el programa para el vector x

[1] Página web sobre vectores `https://www.cplusplus.com/reference/vector/vector/`

[2] Página web sobre algoritmos `https://www.cplusplus.com/reference/algorithm/`

Problemas

1 Encontrar los errores de las siguientes declaraciones de punteros:

```
int x, *p, &y;
char* b= "Cadena larga";
char* c= ‘C’;
float x;
void* r = &x;
```

2 Un array unidimensional se puede indexar con la aritmética de punteros. ¿Qué tipo de puntero habría que definir para indexar un array bidimensional?

3 El código siguiente accede a los elementos de una matriz. Acceder a los mismos elementos con aritmética de punteros.

```
#define N 4
#define M 5
int f,c;
double mt[N][M];
. . .
for (f = 0; f < N; f++)
 {
  for (c = 0; c < M; c++)
   cout << mt[f][c];
  cout << "\n";
 }
```

4 Escribir una función con un argumento de tipo puntero a double y otro argumento de tipo int. El primer argumento se debe de corresponder con un array y el segundo con el número de elementos del array. La función ha de ser de tipo puntero a double para devolver la dirección del elemento menor.

5 Dada las siguientes definiciones y la función gorta:

```
double W[15], x, z;
void *r;
double* gorta( double* v, int m, double k)
{
int j;
for (j = 0; j < m; j++)
if (*v == k)
return v;
return 0,
```

¿Hay errores en la codificación? ¿De qué tipo?
¿Es correcta la siguiente llamada a la función?:

```
r = gorta(W,10,12.3);
```

¿Y estas otras llamadas?:

```
cout << (*gorta(W,15,10.5));
z = gorta(w,15,12.3);
```

6 ¿Qué diferencias se pueden encontrar entre un puntero a constante y una constante puntero?

7 En física se presentan a menudo magnitudes que son el resultado del producto vectorial, por ejemplo:

Torca: $\vec{\tau} = \vec{r} \times \vec{F}$
Aceleración tangencial: $\vec{a}_t = \vec{\alpha} \times \vec{r}$
Fuerza generada por un campo magnético sobre una carga móvil: $\vec{F}_B = q\vec{v} \times \vec{B}$

Para obtener el producto vectorial se puede realizar colocando el primer vector con sus componentes y el segundo vector como una matriz antisimétrica como se muestra a continuación:

$$a = [a_1 \quad a_2 \quad a_3]$$

$$b = [b_1 \quad b_2 \quad b_3]$$

$$\vec{a} \times \vec{b} = [a_1 \quad a_2 \quad a_3] \begin{pmatrix} 0 & -b_3 & b_2 \\ b_3 & 0 & -b_1 \\ -b_2 & b_1 & 0 \end{pmatrix}$$

Realice un programa que calcule el producto vectorial:

a Pasando los parámetros por valor.

b Utilizando punteros a funciones.

c Utilizando un arreglo de funciones.

d Utilizando una estructura que cargue los datos de cada vector.

e Utilizando una estructura que cargue la dirección de la estructura con los datos de cada vector y un parámetro multiplicador del producto vectorial.

f Utilizando un arreglo de punteros a funciones.

g Utilizando sobrecarga de funciones para los items anteriores.

8 Escribir un programa que decida si una matriz de números reales es simétrica. Utilizar

a una función de tipo bool que reciba como entrada una matriz de reales, así como el número de filas y de columnas,y decida si la matriz es simétrica

b otra función que genere la matriz de 10 filas y 10 columnas de números aleatorios de 1 a 100

c un programa principal que realice llamadas a las dos funciones. Nota: usar la transmisión de matrices como parámetros punteros y la aritmética de punteros para la codificación.

9 Codificar funciones que realicen las siguientes operaciones: multiplicar una matriz por un número, y rellenar de ceros una matriz. Nota: usar la aritmética de punteros.

10 Escribir un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapecio y un círculo. Nota: utilizar un array de punteros de funciones, siendo las funciones las que permiten calcular el área.

11 Escribir un programa en C++ que construya un vector de 10 elementos de números enteros aleatorios.

12 Escribir un programa en C++ que construya una matriz de 3x3 elementos de números enteros aleatorios.