Aplicaciones con arreglos y punteros

En este capítulo empezaremos a estudiar los tipos de datos estructurados, en particular, estudiaremos aquí los arreglos.

Para analizar este concepto diremos que un dato simple o estándar es aquel que define un solo elemento, por ejemplo, si definimos una variable como: **int a;** sabemos que en la variable "a" podemos almacenar sólo un valor, a este tipo de variable se le denomina variable estándar.

A diferencia de los datos estándar, los datos estructurados permiten definir variables en las que, en vez de almacenar sólo un valor, podremos almacenar en ellas muchos valores.

La presencia de datos estructurados en los lenguajes de programación se debe a que existen muchos problemas en los que, si sólo se emplearan variables estándar, las soluciones serían muy ineficientes y difíciles de manejar y mantener.

Analicemos los siguientes casos que se pueden presentar:

Supongamos que tenemos un conjunto de datos de los cuales deseamos obtener su desviación estándar. Una fórmula que permite calcular este valor es:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

Esta fórmula nos obliga a calcular previamente el promedio ($\bar{x}$) de todos los datos para luego poder determinar la desviación estándar. Si es cierto que la solución la podemos plantear empleando únicamente variables simples, esta solución no sería eficiente. La razón para esto se debe a que como debemos calcular primero el promedio, tendíamos que leer los datos uno a uno y acumularlos en una suma, como se muestra a continuación:

```
while (true) {
   arch >> dato;
   if(arch.eof())break;
   suma += dato;
   numDat++;
}
prom = (double)suma/numDat;
```

Una vez hecho esto, para poder ahora determinar la desviación estándar debemos calcular otra sumatoria en la que restemos cada dato del promedio, lo elevemos al cuadrado y finalmente se le agregue a la sumatoria. Sin embargo, como ya no tenemos los datos en memoria, no nos va a quedar otra cosa que hacer que volver a leer todos los datos. Esto ya torna ineficiente la solución. Si tuviéramos una forma de almacenar los datos leídos en memoria la primera vez que los leemos, entonces, luego de calcular el promedio, cuando queremos calcular la segunda sumatoria, los datos se tomarían directamente de la memoria y no los tendríamos que leer nuevamente, esto sería muchísimo más rápido y por lo tanto hará más eficiente el proceso.

Otro problema que se presenta y que ilustra muy bien la necesidad del uso de variables estructuradas es el de la determinación de una distribución de frecuencias.

Supongamos que se tiene un archivo en el que se ha registrado la intención de voto de un grupo de personas en una futura elección, supongamos que hay cinco candidatos y cada uno se identifica en el archivo por un número entero entre 1 y 5. A partir de este archivo deseamos saber qué porcentaje quiere votar por el primer candidato, qué porcentaje para el segundo, etc.

Empleando variables simples, la solución podría encaminarse a la definición de cinco variables, las cuales acumularían la cantidad de simpatizantes que votaría por cada candidato. El programa sería más o menos como sigue:

```
int main(void){
  int canad1, canad2, canad3, canad4, canad5;
  int voto, total;
  double porc1, porc2, porc3, porc4, porc5;

  //Inicializamos los contadores
  canad1 = canad2 = canad3 = canad4 = canad5 = 0;
  //Luego de definir una variable de archivo y de abrirlo,
  //leemos los datos y los acumulamos dependiendodel valor
  //leído.
  while (true) {
      arch >> voto;
      if(arch.feof()) break;
      if (voto == 1)
         candid1++;
      else if (voto == 2)
         candid2++;
      else if (voto == 3)
         candid3++;
      else if (voto == 4)
         candid4++;
      else if (voto = 5)
         candid5++;
  }
  total = canad1+canad2+ canad3 + canad4 + canad5;
  porc1 = (double)candid1 / total;
  porc2 = (double)candid2 / total;
  …
}
```

La solución, aunque parece sencilla y adecuada para este caso puntual, no se verá así cuando se quiera extender la solución. Por ejemplo, que la cantidad de candidatos varíe, digamos que en lugar de ser 5 sean 15 candidatos y que una vez modificado el programa se presente otro caso en el que hay sólo 9. Al presentarse estas situaciones no nos quedaría otro camino que modificar el programa para que se adapte al caso y compilarlo nuevamente, y si presenta un nuevo caso volver a hacerlo. Imagínese ahora que hay 150 candidatos.

Un programa se debe plantear de modo que, si las condiciones cambiaran en un futuro, sea el mismo programa el que se adapte a ellas y no tener que hacer modificaciones

que impliquen modificaciones al código y re compilación del programa. Como veremos más adelante esto se puede lograr, en algunos casos empleando tipos de datos estructurados.

Definición:

Un arreglo se define como una colección o conjunto de datos, que tiene como característica que todos los elementos que lo conforman son de un mismo tipo de dato, además todos se identifican por el mismo nombre. La manera de diferenciar un elemento de otro en el conjunto se hace por medio de uno o más índices. Dependiendo del número de índices por los cuales se identifique a los elementos de un arreglo se dirá que este es un arreglo de una dimensión, de dos dimensiones, etc. A un arreglo de una dimensión se le reconoce también con el nombre de "vector", y a uno de dos dimensiones como "matriz".

Implementación de arreglos unidimensionales

Un arreglo se declara de la siguiente manera:

Tipo identificador [Límite];

Donde:

Tipo: indica el tipo de dato que tendrán los elementos del arreglo. Recuerde que todos los elementos del arreglo tienen el mismo tipo de dato.

Límite: indica la cantidad de variables o elementos que tendrá el arreglo.

Ejemplo: int a [10];

Donde "a" es el nombre que identifica al arreglo de tipo int, el cual se define con 10 elementos. Los elementos de este arreglo son:

a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8] y a[9]

Inicialización:

Como cualquier variable, un arreglo puede ser inicializado en el momento de su declaración. Esta operación se realiza de la siguiente manera:

```
int a[10] = {3, 5, 1, 7, 0, 2, 6, 9, 8, 4};
int a[10]   {3, 5, 1, 7, 0, 2, 6, 9, 8, 4};   // Esta forma, sin el =, no es aceptada en C
```

Esto significa que a la variable a[0] se le asigna el valor de 3,a la variable a[1] se le asigna 5, ... a la variable a[9] se le asigna 4.

Si se colocan menos valores de inicialización que la capacidad del arreglo, el resto de elementos del arreglo se inicializa en cero. Así, por ejemplo:

int a[5] = {3, 5, 1}; o int a[5] {3, 5, 1};

Esto significa que: a[0] ⇦ 3, a[1] ⇦ 5, a[2] ⇦ 1, a[3] ⇦ 0, a[4] ⇦ 0.

También, como se muestra a continuación, al definir un arreglo con valores iniciales se puede omitir el tamaño de éste:

int a[ ] = {3, 7, 9, 5}; o int a[ ] {3, 7, 9, 5};

El número de elementos del arreglo lo define la cantidad de valores con el que se desea inicializar. Luego: a[0] ⇦ 8, a[1] ⇦ 7, a[2] ⇦ 9, a[3] ⇦ 5

Sin embargo, se debe tener en cuenta que de alguna manera se debe indicar al compilador el tamaño del arreglo, ya sea colocando un valor entre los corchetes ([ ]) o mediante valores iniciales, es por esto que:

```
int a[ ];  //Es un error grave porque no se
           //le indica el tamaño del arreglo.
```

Algo interesante en el manejo de arreglos es que el índice, que diferencia los elementos, puede ser reemplazado por una expresión cuyo valor esté dentro del rango definido para los índices del arreglo. En los siguientes ejemplos se muestra esta característica.

```
int a[5], n = 2 , p = 15;
a[2] = 35;
a[0] = p*3;    // asigna 45 (p*3 = 15*3 = 45) al primer elemento del
               // arreglo (a[0])
a[n + 1] = 17;  // asigna 17 al cuarto elemento del arreglo, esto
                //esn+1 = 2 + 1 =3
a[p/3-1] = 23*n/5;    // asigna 9 (23*n/5= 9) al quinto elemento
                      //del arreglo (p/3-1 = 4)
```

La tarea de asignar valores a los elementos de un arreglo, como se ha podido ver, es una tarea sencilla, sin embargo, debido a que el compilador de C/C++ no verifica los rangos para los que se ha definido el arreglo, si no se toma el cuidado necesario se podría incurrir en errores muy serios.

Cuando uno escribe una orden como "a[n] = 3", el sistema calcula la dirección donde se colocará el dato, de la siguiente manera:

$$DMA_n = DMA + n \times sizeof(TA)$$

Dónde:

DMA	es la dirección de memoria donde empieza el arreglo "a".
DMA_n	es la dirección de memoria del elemento n del arreglo "a".
sizeof(TA)	es el tamaño del tipo de dato del arreglo "a".

Este cálculo lo realiza el sistema a ciegas, es decir que no verifica si el valor dado como n sea positivo o negativo o si es más grande que el número de elementos definidos para el arreglo.

El siguiente ejemplo muestra claramente esta afirmación.
Ejemplo:

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

int main(int argc, char** argv) {
    int a=10, b=20, c=30, d[5], e=40, f=50,g=60;
    cout<<"Direcciones de memoria de las variables:" << endl;
    cout<<"A="<<&a<<" B="<<&b<<" C="<<&c<<" D="<<d<<" E="<<&e<<" F="<<&f<<" G="<<&g<<endl<<endl;
    cout<<"Valores iniciales de las variables:"<<endl;
    cout<<"A="<<a<<" B="<<b<<" C="<<c<<" E="<<e<<" F="<<f<<" G="<<g<<endl<<endl;
    d[-1]=22;
    cout<<"Asignamos 22 a d[-1]"<<endl;
    cout<<"A="<<a<<" B="<<b<<" C="<<c<<" E="<<e<<" F="<<f<<" G="<<g<<endl<<endl;
    d[-2]=77;
    cout<<"Asignamos 77 a d[-2]"<<endl;
    cout<<"A="<<a<<" B="<<b<<" C="<<c<<" E="<<e<<" F="<<f<<" G="<<g<<endl<<endl;
```

```
    d[6]=55;
    cout<<"Asignamos 55 a d[6]"<<endl;
    cout<<"A="<<a<<" B="<<b<<" C="<<c<<" E="<<e<<" F="<<f<<" G="<<g<<endl<<endl;
    d[7]=99;
    cout<<"Asignamos 99 a d[7]"<<endl;
    cout<<"A="<<a<<" B="<<b<<" C="<<c<<" E="<<e<<" F="<<f<<" G="<<g<<endl<<endl;

    return 0;
}
```

Al ejecutar este programa obtendremos como resultado algo similar a esto:

```
Direcciones de memoria de las variables:
A=0x61fefc B=0x61fef8 C=0x61fef4 D=0x61fee0 E=0x61fedc F=0x61fed8 G=0x61fed4

Valores iniciales de las variables:
A=10 B=20 C=30 E=40 F=50 G=60

Asignamos 22 a d[-1]
A=10 B=20 C=30 E=22 F=50 G=60

Asignamos 77 a d[-2]
A=10 B=20 C=30 E=22 F=77 G=60

Asignamos 55 a d[6]
A=10 B=55 C=30 E=22 F=77 G=60

Asignamos 99 a d[7]
A=99 B=55 C=30 E=22 F=77 G=60
```

Observe las direcciones de memoria asignadas a cada variable, en la versión en que fue ejecutado el programa, los entero se almacenan en cuatro bytes, haciendo cálculos muy sencillo se puede dar cuenta del porqué de esos resultados.

Es por esta razón que se debe tener mucho cuidado cuando asigna valores a los elementos de un arreglo. Si se sale del rango de elementos para los que fue definido el arreglo, si bien es cierto que el compilador no lo hará notar, el programa puede dar resultados incoherentes.

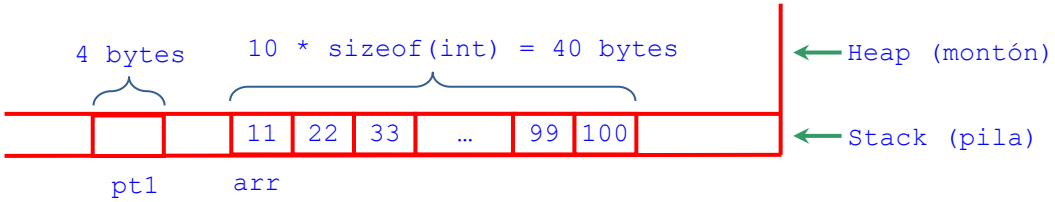
Punteros como arreglos:

Un puntero, a pesar que su definición es muy diferente que un arreglo, su naturaleza es la misma. En siguiente ejemplo muestra esta condición.

```
int arr[10]  {11, 22, 33, 44, 55, 66, 77, 88, 99, 100};
int *pt1, *pt2;

cout<<"Usando arr: "<<endl;
for (int i=0; i<10; i++)
    cout<<setw(4)<< arr[i];
cout<<endl;

pt1 = arr; <=== NO HAY ERROR

cout<<"Usando pt1:   ");
for (int i=0; i<10; i++)
    cout<<setw(4)<< pt1[i];
cout<<endl;

arr = pt1; <=== ESTO SI ES UN ERROR
```

El ejemplo nos muestra que un puntero y un arreglo son de la misma naturaleza, por lo que se pueden manejar de la misma manera. Sin embargo, hay varias diferencias entre un arreglo y un puntero.

La primera de ella son los espacios de memoria asignados, mientras que el puntero se almacena siempre en un espacio de memoria de 4 bytes, el arreglo se almacena dependiendo del tipo en que fue definido y de la cantidad de elementos que se le ha definido.



La segunda es que el puntero es una variable a la que se le puede asignar cualquier dirección de memoria durante la ejecución del programa, el arreglo, como veremos luego, también es un puntero, pero la dirección de memoria se le asigna en el momento de la compilación y esa dirección es constante, por eso la última línea del ejemplo produce un error.

Arreglos como parámetros de funciones:

Cuando se emplean arreglos como parámetro de una función, la dirección de inicio del arreglo es enviada como parámetro y no, como se puede suponer, todo el arreglo. Por lo tanto, cualquier modificación que se haga sobre los elementos del arreglo en la función afectará a los elementos del arreglo en la función que lo llamó.

El siguiente ejemplo muestra lo que indicamos:

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

int main(int argc, char** argv) {
    int arr[5] = {1,2,3,4,5}, nd=5;
    modificaArreglo(arr, nd);
    imprimeArreglo(arr, nd);
    return 0;
}

// Definición de funciones:
void modificaArreglo(int *, int);
void imprimeArreglo (int *, int);

// Implementación de funciones:
void modificaArreglo(int *arr, int nd){
    arr[1] = 77;
    arr[3] = 313;
}

void imprimeArreglo (int *arr, int nd){
    for(int i = 0; i < 5; i++)
        cout<<setw(5)<<arr[i];
    cout<<endl;
}
```

Al ejecutar este programa obtendremos como resultado esto: `1 77 3 313 5`

Por otro lado, cuando se elabore una función, se debe tener muy claro qué es lo que se necesita enviar como parámetro.

A continuación, enumeramos las diferentes formas en que se puede emplear un arreglo como parámetro a una función:

Dado un arreglo definido como: `int arr[10];`

TAREA QUE SE DESEA REALIZAR	FORMA DE LLAMAR A LA FUNCIÓN	FORMA DE DEFINIR O IMPLEMENTAR LA FUNCIÓN
Si se desea usar o modificar los elementos de un arreglo en una función.	`f(arr);`	`void f(int *arr){…`
Si se desea usar un elemento de un arreglo en una función.	`f(arr[i]);`	`void f(int a){…`
Si se desea modificar un elemento de un arreglo en una función.	`f(arr[i]);`	`void f(int &a){…`

Algebra de punteros:

Las direcciones de memoria se pueden manipular, realizando operaciones con ellas. Las operaciones que se pueden realizar son la suma y resta, sin embargo, la interpretación de estas operaciones no son las mismas que cuando se aplican con valores numéricos.

Cuando se define un puntero como: int *pt; luego de asignarle al puntero una dirección válida, la siguiente expresión: pt + 2 no quiere decir que a la dirección que contiene pt se le va a sumar el valor de 2. Esa expresión quiere decir que a la dirección de memoria se le va a 2 multiplicado por el tamaño de la variable referenciada por el puntero, en este caso al ser un puntero de tipo int lo que se va a sumar es:

2*sizeof(int) = 2 * 4 = 8.

Donde sizeof es un operador definido en el lenguaje C++ que devuelve el tamaño en bytes de un tipo de dato o variable.

Esto se hace para facilitar operaciones que se realicen con los punteros, el siguiente ejemplo aclara este concepto:

```
int main(int argc, char** argv) {
    int arr[10]={11, 22, 33, 44, 55, 66, 77, 88, 99, 100};
    int *pt1, *pt2;

    pt1 = arr;

    cout <<left<<setw(14)<<"Usando pt1:";
    for (int i=0; i<10; i++)
        cout <<right<<setw(4)<<pt1[i];
    cout<<endl<<endl;

    // Algebra de punteros:
    pt2 = pt1 + 1; //<============
    cout <<left<<setw(14)<<"Pt2 ="<<right<<setw(4)<<*pt2<<endl;

    cout <<left<<setw(14)<<"Todo   pt2:";
    for (int i=0; i<10; i++)
        cout <<right<<setw(4)<<pt2[i];
    cout<<endl<<endl;

    // Otra forma de manipular los punteros
    cout <<left<<setw(14)<<"Otra forma:";
    for (int i=0; i<10; i++)
        cout <<right<<setw(4)<<*(pt1 + i);  //<============
    cout<<endl<<endl;


    return 0;
}
```

El resultado al ejecutar el programa se muestra a continuación:

```
Usando pt1:     11  22  33  44  55  66  77  88  99 100
Pt2 =           22
Todo   pt2:     22  33  44  55  66  77  88  99 1006356728
Otra forma:     11  22  33  44  55  66  77  88  99 100
```

En el programa se observa primero que el puntero pt1 recibe la dirección del primer byte del espacio separado para el arreglo. Luego, bajo el comentario "//Algebra de punteros" se ejecuta la orden: pt2 = pt1 + 1; esto, como se indicó se interpreta como:

pt2 = pt1 + 1*sizeof (int);

Por lo tanto, pt2 recibirá la dirección del segundo elemento del arreglo y no del segundo byte del arreglo, con lo que pt2 está listo para manipular sin ningún problema el segundo elemento del arreglo.

Debe entender que si se incrementara solo en 1 la dirección de tp2 el resultado sería incoherente, porque la variable referenciada estaría compuesta por los tres últimos bytes del primer elemento del arreglo y el primer byte del segundo elemento.

Finalmente observe que la manipulación de un arreglo se puede hacer aplicando el álgebra de punteros.

Aplicaciones que emplean arreglos de una dimensión:

Volvamos al problema de la desviación estándar, queremos ahora elaborar un programa que solucione el problema. Aquí se debe tomar en cuenta que cada vez que se ejecute el programa la muestra que se emplee para el cálculo será diferente. No se puede suponer que siempre las muestras serán las mismas y del mismo tamaño. Entonces, debido a esto es que el tamaño del arreglo debe estar definido antes de ejecutar el programa y que no es lógico que el código del programa tenga que ser modificado en cada ejecución para adaptarse a la muestra. Debemos definir el tamaño máximo de la muestra, el cual debe ser analizado de acuerdo a las condiciones del problema.

Desviación estándar: En este primer caso queremos calcular el promedio y la desviación estándar referidas a las notas de un horario de un curso en diferentes momentos. Entonces podemos determinar que, pensando en la universidad, donde no hay salones de más de 70 carpetas en sus aulas, podríamos fijar el número máximo de las muestras en 100, tomando un factor de seguridad. Este valor será asumido para determinar el tamaño del arreglo que usemos en nuestro programa.
Entonces, el programa que solucionará este problema será el siguiente:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "FuncEstadisticas.h"

int main(int argc, char** argv) {
    int nota[100], numNot;
    double prom, desvEst;

    leerNotas(nota, numNot);
    prom = promedio(nota, numNot);
    desvEst = desvEstandar(prom, nota, numNot);
    cout<<setprecision(3)<<fixed;
    cout<<left<<setw(22)<<"Promedio:"<<right<<setw(10)<<prom<<endl;
    cout<<left<<setw(22)<<"Desviacion Estandar:"<<right<<setw(10)<<desvEst<<endl;

    return 0;
}
```

La definición de las funciones empleadas en el programa se muestra a continuación:

```
#ifndef FUNCESTADISTICAS_H
#define FUNCESTADISTICAS_H
    void leerNotas(int *, int &);
    double promedio(int *, int);
    double desvEstandar(double, int *, int);
#endif /* FUNCESTADISTICAS_H */
```

La implementación de estas funciones las vemos a continuación:

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include <cmath>
#include "FuncEstadisticas.h"

void leerNotas(int *nota, int &numNot){
    ifstream archNotas("notas.txt", ios::in);
    if(not archNotas.is_open()){
        cout<<"ERROR: No se puede abrir el archivo de notas.txt"<<endl;
        exit(1);
    }
    int nta;

    numNot = 0;
    while(1){
        archNotas >> nta;
        if (archNotas.eof())break;
        nota[numNot] = nta;
        numNot++;
    }
}

double promedio(int *nota, int numNot){
    int suma = 0;
    for (int i = 0; i < numNot; i++)
        suma += nota[i];

    return (double)suma/numNot;
}

double desvEstandar(double prom, int *nota, int numNot){
    double suma = 0;
    for (int i=0; i<numNot; i++)
    suma += pow(nota[i]-prom, 2);// <====== Requieren cmath.h
    return sqrt(suma/(numNot-1));// <======
}
```

Distribución de frecuencias: Como segundo caso analicemos el problema que se presenta con una elección de candidatos. Este problema también se conoce como distribución de frecuencias.

Para resolverlo se cuenta con dos archivos como los que se muestran a continuación:

Candidatos.txt

```
757575
122112
543731
849000
123456
645936
```

Elección.txt

```
5  2  5  1  6  5  5
6  5  5  5  6  2  2  4  3
5
2  6  1  1  5  2
5  5  5  5  4  5  2  5  5  5  5
6  2  2  5
5  6  5  2  5  1  3  2  6  5  5
…
```

El archivo de candidatos contendrá la identificación de los candidatos (valores enteros) ordenados por el número que lo identifica en la cédula de votación, esto es el candidato 757575 se identifica en la cédula con el número 1, el 122112 con el número 2, etc. El otro archivo tendrá los datos obtenidos en la elección, los cuales serán simplemente el número del candidato por el que votaría el ciudadano.

De acuerdo a esto, el programa leerá los datos del archivo de candidatos y los almacenará en un arreglo, la lectura nos permitirá saber también cuántos candidatos tenemos, de modo de poder validar los votos. Luego de esto se leerán uno a uno los votos para realizar el conteo por cada candidato. El reporte final mostrará los candidatos seguidos del número de votos que consiguió, el porcentaje del total de votos y una gráfica sencilla que facilite el poder ver quién ganó la elección.

El programa que solucionará este problema será el siguiente:

```
#include "FuncionesAuxiliares.h"
#define MAX_CAND 20

int main(int argc, char** argv) {
    int cand[MAX_CAND],votos[MAX_CAND] {}, numCand=0;
    double porcentaje[MAX_CAND];

    leerCandidatos(cand,numCand);
    contarVotos(votos,numCand);
    calcularPorcentajes(votos,numCand,porcentaje);
    imprimirReporteConResultadosDeEleccion(cand,votos,porcentaje,numCand);

    return 0;
}
```

Note que se han definido tres arreglos, uno para guardar la identificación de los candidatos, otro para contar los votos de cada candidato y el tercero para guardar los porcentajes que obtuvo cada uno. Note también que la cantidad de votantes pueden ser miles o millones, por lo que no tiene sentido y sería un desperdicio de recursos guardar esos miles o millones de datos en un arreglo, hay que pensar qué es lo que realmente se requiere almacenar. La distribución de frecuencias busca saber cuántas personas votaron por un determinado candidato, por lo tanto, el programa debe contar los votos de cada candidato. De acuerdo a esto se requiere un contador por cada candidato y por consiguiente se necesita solo de un arreglo que cumpla esta función.

La definición de las funciones empleadas en el programa se muestra a continuación:

```
#ifndef FUNCIONESAUXILIARES_H
#define FUNCIONESAUXILIARES_H

    void leerCandidatos(int *,int &);
    void contarVotos(int *, int);
    void calcularPorcentajes(int *,int,double *);
    void imprimirReporteConResultadosDeEleccion(int *,int*,double*,int);
    int calculaMaximo(int *, int); //Ver más adelante

#endif /* FUNCIONESAUXILIARES_H */
```

La implementación de estas funciones las vemos a continuación:

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "FuncionesAuxiliares.h"
#define MAX_CAR 50  // Ver función imprimirReporteConResultadosDeEleccion

void leerCandidatos(int *cand, int &numCand){
    ifstream archCand("candidatos.txt",ios::in);
    if(not archCand.is_open()){
        cout<<"ERROR: Nose pudo abrir el archivo candidatos.txt"<<endl;
        exit(1);
    }
    int candidato;

    numCand = 0;
    while(1){
        archCand >> candidato;
        if (archCand.eof())break;
        cand[numCand]= candidato;
        while(archCand.get() != '\n');
        numCand++;
    }
}

void contarVotos(int *votos, int numCand){
    ifstream archVotos("eleccion.txt",ios::in);
    if(not archVotos.is_open()){
        cout<<"ERROR: Nose pudo abrir el archivo eleccion.txt"<<endl;
        exit(1);
    }
```

```
    int voto;

    // El arreglo de votos ya fue inicializado: (votos[MAX_CAND] {})

    // Leemos el voto y de acuerdo a él, incrementamos el contador del candidato.
    while(1){
        archVotos>>voto;
        if (archVotos.eof())break;
        if (voto>=1 && voto<=numCand) votos[voto-1]++;
    }
}
```

En la función ***contarVotos*** se puede apreciar algunas de las ventajas del uso de los arreglos, aquí se ha considerado un arreglo en el que cada elemento llevará la cuenta de los votos de cada candidato. En este caso, a diferencia de lo que hicimos en el ejemplo al inicio de este capítulo, no se requiere que luego de leer la intención de voto de un elector tengamos que ejecutar una condicional por cada candidato para saber qué variable debemos incrementar, tampoco necesitamos realizar todo un proceso de búsqueda. Aquí haremos que el índice de los elementos del arreglo coincida con el número del candidato, entonces, luego de leer la intención de voto, ésta se usará como índice del elemento del arreglo que queremos incrementar. De este modo, sin importar cuántos candidatos haya, sólo se requiere una instrucción para realizar esta labor.

```
void calcularPorcentajes(int *votos, int numCand, double *porcentaje){
    int suma=0;
    for (int v=0; v<numCand; v++)
        suma += votos[v];       // Determinamos el número de votantes
    for (int v=0; v<numCand; v++)
        porcentaje[v] = (double)votos[v]/suma*100;
}
```

En la función que mostrará el reporte, queremos introducción una gráfica sencilla mediante un histograma de barras que muestres como quedó finalmente la elección.

Como se trata precisamente de un gráfico de barras, y en archivo de salida sólo se pueden colocar caracteres, las barras se dibujarán empleando el caracter '*', al colocar varios de estos caracteres seguidos se verá como una barra: '**********'.

Según esto último y en vista que estamos hablando de miles o millones de votantes, cada candidato podría acumular cientos, miles o millones de votos. Esto nos hacer ver que no podemos imprimir un caracter '*' por cada voto, sino que cada caracter debe representar un conjunto de votos. El problema es que no podemos dar un valor fijo a este conjunto, porque la muestra de votantes puede cambiar mucho. Por lo tanto, lo que se ha decidido es determinar el tamaño de ese conjunto de manera variable, en función de la muestra.

Por lo tanto, podemos establecer que las barras tendrán una zona de aproximadamente 50 caracteres como máximo para mostrarse. Es por eso que se define, al inicio del archivo d implementación, una constante simbólica denominada MAX_CAR que se usará para determinar el largo de cada barra.

Esto quiere decir que luego de calcular la cantidad de votos que tiene cada candidato, se determina quién tiene más votos, luego se establece que esa cantidad será graficada en el área total destinada para las barras, esto es 50 caracteres. Las otras cantidades, que son menores, tendrán un tamaño proporcional al mayor. Será cuestión de aplicar una regla de tres simple.

```
void imprimirReporteConResultadosDeEleccion(int *cand, int*votos, ouble*porcentaje,
                                            int numCand){
    ofstream archRep("reporteResultadosDeEleccion.txt",ios::out);
    if(not archRep.is_open()){
        cout<<"ERROR: Nose pudo abrir el archivo reporte.txt"<<endl;
        exit(1);
    }
```

```
        int maxVotos, numCar;

        maxVotos = calculaMaximo(votos,numCand);
        archRep<<setw(10)<<"Candidato"<<setw(10)<<"Votos"<<setw(8)<<"%"<<endl;
        archRep.precision(2);
        archRep<<fixed;
        for (int v=0; v<numCand; v++){
            archRep << setw(10)<<cand[v]<<setw(10)<<votos[v]<<setw(10)
                    <<porcentaje[v]<<"% |";
            numCar = votos[v]*MAX_CAR/maxVotos;
            for(int c=0; c<numCar; c++) archRep.put('*');
            archRep<<endl;
        }
    }

    int calculaMaximo(int *votos, int numCand){
        int max=0;
        for(int v=0; v<numCand; v++)
            if(votos[v] > max)  max = votos[v];
        return max;
    }
```

La ejecución del programa mostrará el siguiente reporte:

```
 Candidato     Votos       %
  46115231       770     7.61% |********
  22358262      2279    22.54% |************************
  72883328       987     9.76% |**********
  15723944       277     2.74% |***
  65527915      4309    42.61% |**************************************************
  39006077      1490    14.73% |***************
```