

Programación

Unidad 7: Vectores y Estructuras

“Los malos programadores se preocupan por el código. Los buenos programadores se preocupan por las estructuras de datos y sus relaciones.”

~ Linus Torvalds

Vectores y Estructuras

En esta unidad nos proponemos presentarte nuevas estructuras de datos para almacenar varios elementos. Te explicaremos qué son los vectores y las matrices y cómo utilizarlos. Además, te mostraremos qué son las Estructuras de Datos.

En las unidades anteriores trabajamos almacenando los datos de los programas en variables y constantes. Estos identificadores tienen la particularidad de que almacenan un único valor. Las variables y constantes de tipos de datos simples son muy útiles, pero hay algoritmos más complejos en los que necesitarás guardar múltiples datos. Es por eso que serán necesarias otras estructuras de datos para poder trabajar con muchos valores y de distintos tipos de datos.

En esta unidad, como anticipamos, veremos cómo trabajar con vectores, matrices y estructuras de datos

Vectores

Supongamos que queremos calcular el promedio de edades de los alumnos de un curso. Seguramente no tendrás ningún problema en resolverlo, ya que deberás realizar un programa en el que el usuario vaya ingresando las edades, y por cada edad ingresada se deberá acumular y contar. De esta manera tendrás la suma total de las edades y la cantidad de alumnos, datos necesarios para obtener el promedio.

Ahora bien, le agregaremos un resultado más: queremos saber cuántos alumnos superan la edad promedio.


¿Cómo realizarlo?

Una vez que obtuviste el promedio de edades, deberás comparar cada edad para determinar si supera la edad promedio, y en caso afirmativo, contarla como una edad que supera la edad promedio.

Pero, ¿Cómo hacerlo si los datos que ingresó el usuario ya no los tenés? ... no los pudiste guardar porque no sabías la cantidad de alumnos, puede ser 10, 50 o un curso de 500.

Si la cantidad de alumnos es, por ejemplo, 500:

¿Utilizarás 500 variables para almacenar los datos de 500 alumnos?



Te estarás imaginando que esta no es una solución práctica. Por lo tanto necesitarás alguna otra estructura de datos para poder almacenar todas las edades de los alumnos y poder consultarlas nuevamente. Es en este momento donde te presentamos a los arreglos.

¿Qué es un vector?

Un vector puede guardar varios datos asociados a un mismo nombre. Es una variable que, en lugar de tener un único valor, tiene asociado una cantidad finita de valores. ¿Y por qué finita? Porque al crear un arreglo debés indicar cuál es la cantidad de elementos que vas a guardar en él.

Los vectores también se denominan arreglos unidimensionales o arrays.

Veamos un ejemplo....

Para definir un arreglo, como te contamos antes, necesitás conocer la cantidad de elementos que tendrá. De ahí surge que un arreglo es una estructura de datos estática, ya que siempre tiene la misma cantidad de elementos.

Declaración de un vector

```
int vec[4];
```

Como dijimos antes un vector es una estructura que nos permite con el nombre de una variable acceder a múltiples valores almacenados, en este ejemplo sencillo declaramos un vector llamado 'vec' que podrá guardar hasta cuatro valores de tipo entero.

Podemos definir un vector de la siguiente manera:

Es un conjunto finito de elementos del mismo tipo de dato y naturaleza que ocupan posiciones contiguas de memoria y se encuentran asociados a un mismo nombre en común.

Analicemos la definición de arreglo...

- Conjunto finito: La cantidad de elementos de un arreglo es finita y se conoce en el momento de definir el arreglo. No se pueden agregar o eliminar elementos, ya que un arreglo tiene siempre la misma cantidad de componentes.
- Elementos del mismo tipo de dato y naturaleza: Todos los elementos son del mismo tipo de dato. Por ejemplo, si vamos a utilizar un arreglo para guardar edades de personas, todos los elementos serán del tipo de dato ENTERO

- Ocupan posiciones contiguas de memoria: Los elementos se almacenan en la memoria uno a continuación de otro. Como cada uno de los elementos ocupa el mismo espacio, se puede acceder a cualquier elemento conociendo la posición de memoria del primer elemento.
- Asociados a un mismo nombre: Un arreglo es una VARIABLE, por lo tanto, tendrá un solo nombre que estará asociado a un conjunto de valores. A cada uno de los elementos se accede directamente a través de su posición, utilizando lo que llamaremos el Índice.

Declaración de vectores

Para poder declarar un vector es necesario conocer la cantidad de elementos máxima que tiene que almacenar, esto también lo llamamos dimensión y es el valor que se pasa entre corchetes.

```
int vec[4]
```

Pero podría pasar que esa cantidad sea variable según algún dato ingresado por el usuario o calculado por el algoritmo, con lo cual tendremos que usar una variable en la declaración que nos dirá cuál es la dimensión:

```
int vec[N]
```

Lo importante y que no se debe olvidar nunca es que **la variable N debe tener el valor** al momento de usarla en la declaración del vector, es decir lo este código es **incorrecto**:

```
1 int main() {
2     int N;
3     int edades[N]; //Define el vector de N elementos pero N no tiene valor!!!
4     cout << "Ingrese la cantidad de edades";
5     cin >> N;
6 }
7
```

Ya que N no tiene el valor ingresado en la línea 3 al momento de la declaración del vector. La forma correcta sería declarar el vector luego de que se lee el valor en N:

```
1 int main() {
2     int N;
3     cout << "Ingrese la cantidad de edades";
4     cin >> N;
5
6     int edades[N];
7 }
```

También es posible declarar e inicializar el vector utilizando cualquiera de las siguientes dos formas:

1	<code>int edades[] = {1, 2, 3, 4};</code>	<code>int edades[4] = {1, 2, 3, 4};</code>
---	---	--

En la primera no es necesario especificar la dimensión dado que se infiere de la cantidad de elementos usados en la inicialización.

Acceso a cada elemento

Índice

Como dijimos previamente, todos los elementos del vector están asociados a un mismo nombre, pero tenemos una manera de acceder a cada uno de los elementos de forma individual, y eso se hace a través del índice.

Definición	El índice es la posición relativa que tiene cada elemento en el arreglo .
-------------------	---

Posición inicial

En C/C++ y en todos los lenguajes de programación el primer elemento se encuentra en la posición 0, es decir que para acceder al primer elemento debemos acceder a la posición 0; para acceder al segundo elemento accederemos a la posición 1 y así sucesivamente.

Acceso directo

Se puede acceder directamente a cada elemento particular por medio del índice sin necesidad de tener que pasar por los elementos anteriores. Esto es lo que llamamos acceso directo a cada elemento.

Para especificar la posición del vector a la que queremos acceder, tanto para leer como para guardar un valor utilizaremos su posición encerrada entre corchetes []:

<code>edades[POS]</code>

Donde POS es la posición del elemento al que queremos acceder. Recordemos que se tienen que cumplir las siguientes condiciones:

POS >= 0 Y POS < N

Veamos un ejemplo para entender el acceso a cada elemento

1	<code>edades[0] = 20;</code>
2	<code>edades[1] = 21;</code>
3	<code>cout << edades[1];</code>

1. En la primera acción asignamos el valor 20 al elemento de la posición 0.
2. En la segunda acción asignamos el valor 21 al elemento de la posición 1
3. En la tercera acción mostramos el valor de la posición 1.

¿Qué valor se mostrará por pantalla?

El número **21**.

Cuando trabajamos con vectores es fundamental poder identificar la posición y el valor del vector en la posición. El elemento está relacionado por la posición que ocupa (el índice), mientras que el valor es el contenido (el dato) que se encuentra almacenado en esa posición.

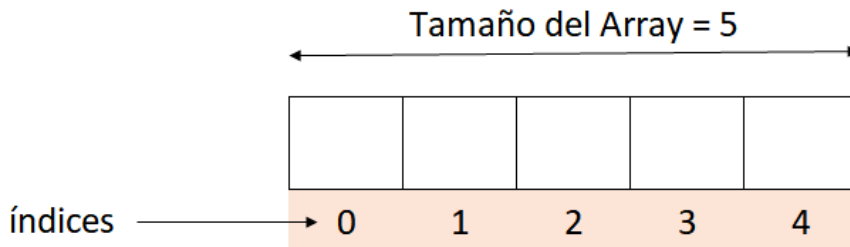
En el ejemplo tenemos que el elemento de la posición 0 tiene el valor 20, y el elemento de la posición 1 tiene el valor 21.

Representación gráfica

Siguiendo con el ejemplo del vector de edades, si tuviéramos la siguiente declaración:

```
1 int edades[5];
```

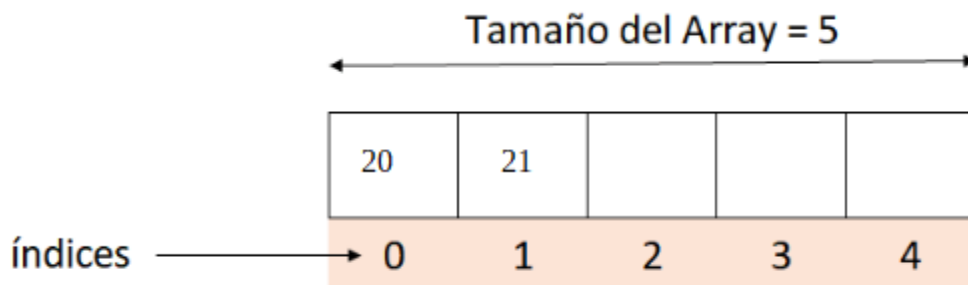
Gráficamente el arreglo definido lo podemos ver de la siguiente manera:



Luego de las sentencias:

```
1 edades[0] = 20;  
2 edades[1] = 21;
```

Resulta



En este ejemplo se puede decir que:

- Edades sub 1 es igual a 21.
- El valor del elemento de la posición 1 es 21.
- Si bien tenemos 5 elementos en EDADES, al haber cargado valores para los dos primeros, diremos que el arreglo tiene 2 elementos por lo cual hay 3 posiciones “libres”. Se debe tener especial cuidado con las posiciones libres ya que si no han sido inicializadas esas posiciones podrían tener basura.
- Las posiciones con datos válidos son la 0 y la 1, por lo cual esas son las posiciones que accederemos en caso que queramos leer valores.

Algoritmos básicos: Cargar Vector

Veamos un algoritmo que utilizaremos para cargar un arreglo con los datos que ingresa un usuario por teclado

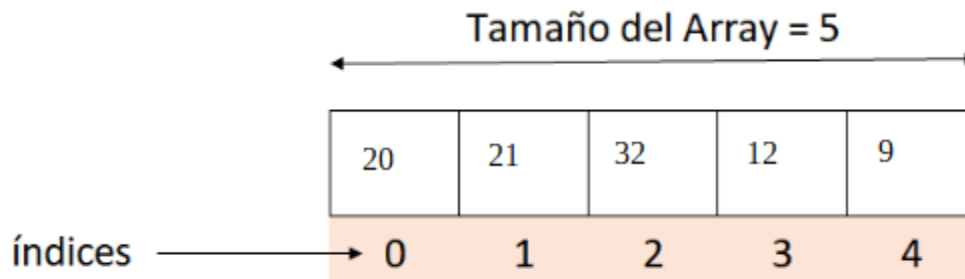
```
1 int main() {
2     int N = 0;
3     cout << "Ingrese cantidad de edades";
4     cin >> N;
5     int edades[N];
6     for(int i = 0; i < N; i++) {
7         cout << "Ingrese una edad: ";
8         cin >> edades[i];
9     }
}
```

Analicemos el algoritmo:

* Lo primero que hacemos es averiguar la cantidad de elementos que deberemos almacenar. En este caso esa cantidad es provista por el usuario.

* Luego declaramos el vector de **N elementos**, tener en cuenta que **N ya debe tener un valor al realizar la declaración del vector**.

* Por último generamos un ciclo que repetirá las sentencias para que el usuario ingrese una edad. En este caso queremos llenar el vector completo por lo cual el contador del for irá de 0 a N – 1. De esta forma en la primer iteración el valor ingresado será almacenado en `edades[0]` y el último en `edades[N – 1]`. Por ejemplo, si el usuario ingresara un `N = 5` y luego las edades 20, 21, 32, 12 y 9 el vector al finalizar el algoritmo quedaría:



Algoritmos básicos: Mostrar Vector

Ahora veremos una forma de mostrar todos los elementos junto con la posición en que se encuentran dentro del vector. Este algoritmo lo podremos utilizar cada vez que necesitemos mostrar todo un vector.

```
1  int main() {
2      int N = 0;
3      cout << "Ingrese cantidad de edades";
4      cin >> N;
5      int edades[N];
6      for(int i = 0; i < N; i++) {
7          cout << "Ingrese una edad: ";
8          cin >> edades[i];
9      }
10     for(int i = 0; i < N; i++) {
11         cout << "Posicion: " << i << " Valor: " << edades[i] ;
12     }
13 }
```

En este algoritmo lo que vemos que se ha agregado es otro ciclo que vuelve a generar las posiciones del vector de 0 a $N - 1$ y por cada iteración imprimirá la posición y el valor del vector en la posición. La salida de este algoritmo si cargara los mismos datos antes enumerados sería:

```
>Posicion: 0 Valor: 20
>Posicion: 1 Valor: 21
>Posicion: 2 Valor: 32
>Posicion: 3 Valor: 12
>Posicion: 4 Valor: 9
```

Arreglos como parámetros

Como ya sabemos, los parámetros pueden ser pasados por Valor o por Referencia. Pero si desearamos pasar un vector por valor esto requeriría que se nos asigne nuevamente la cantidad de memoria necesaria para el vector que ya pedimos en el algoritmo principal, y además debería hacer el copiado de todos los bytes de un espacio al otro. Para evitar todo esto, lo que haremos es pasar los vectores por referencia SIEMPRE. De hecho C/C++ no permite otra forma de pasaje de vectores como parámetro.

Para especificar que un módulo recibe un vector, y nuevamente remarcamos lo que recibirá es la dirección de memoria del vector ya que es pasado por referencia, lo que haremos es especificar el tipo del vector, el nombre con el que lo referenciamos en el módulo y los corchetes vacíos [] para indicar que es un vector.

```
1 void funcion (int vector[]) {
2 }
```

Ahora pensemos, si quisiera en este procedimiento imprimir todos los elementos del vector, me encontraría con el problema de saber cuántos elementos hay. Por lo tanto además del vector siempre deberemos recibir la cantidad de elementos presentes para poder realizar el ciclo correctamente.

Veamos por ejemplo el procedimiento imprimir que haga lo mismo que teníamos antes en el algoritmo principal:

```
1 void imprimir(int vec[], int cant) {
2     for(int i = 0; i < cant; i++) {
3         cout << "Posicion: " << i << " Valor: " << vec[i];
4     }
5 }
```

De esta forma el procedimiento recibe la referencia al vector por lo cual ya tiene los valores cargados y podrá recorrerlo e imprimirlo.

También podríamos hacer otro procedimiento que realice la carga:

```
1 void cargar(int vec[], int cant) {
2     for(int i = 0; i < cant; i++) {
3         cout << "Ingrese una edad";
4         cin >> vec[i];
5     }
6 }
```

Ahora nuestro programa principal podría ser:

```
1 int main() {
2     int N;
3     cout << "Ingrese la cantidad de edades";
4     cin >> N;
5     int edades[N];
6     cargar(edades, N);
7     imprimir(edades, N);
8 }
```

¿Cómo usamos los vectores?

El objetivo de utilizar los vectores o las distintas estructuras es permitirnos almacenar muchos datos de forma simple para su acceso. Con esto además el conjunto de datos puede separarse del algoritmo de su obtención y tratamiento. Es decir antes al solicitar las edades si queríamos saber cuál era la mayor debíamos averiguarlo en la misma carga de datos, ya que los datos luego no los tendríamos más. Esto generaba una mezcla de etapas como hemos dicho al inicio del apunte, todo algoritmo tiene entradas, procesos y salidas, pero en este caso debíamos hacer la entrada junto con algo de proceso.

Ahora podemos separar de forma más clara los procesos de las entradas, y esto redundará en una mejor modularización de nuestros algoritmos.

Por ejemplo el programa que pide N edades y averigua la mayor ahora lo podremos escribir usando una función buscaMax:

```
1 int buscaMax( int vec[], int cant) {
2     int max = vec[0];
3     for(int i = 1; i < cant; i++) {
4         if (vec[i] > max) {
```

```

5         max = vec[i];
6     }
7 }
8     return max;
9 }

```

Si analizamos el algoritmo es lo mismo que hubiéramos hecho antes de conocer los vectores, tomar el primer valor ingresado como máximo y luego por cada valor evaluar si es mayor que el máximo. La diferencia ahora es que para tomar el primer valor, usaremos el índice 0 y luego recorreremos el vector desde 1 (ya que el 0 lo tenemos como máximo) hasta la última posición del vector.

Otra alternativa mejor sería devolver no el valor máximo sino la posición del máximo, ya que con la posición se podrá acceder al valor:

```

1  int buscaPosMax( int vec[], int cant) {
2      int max = vec[0];
3      int pos = 0;
4      for(int i = 1; i < cant; i++) {
5          if (vec[i] > max) {
6              max = vec[i];
7              pos = i;
8          }
9      }
10     return pos;
11 }

```

Otra función que podremos realizar ahora es promedio que reciba un vector de enteros y devuelva el promedio de sus elementos:

```

1  float promedio(int vec[], int n) {
2      int sum = 0;
3      for(int i = 0; i < n; i++) {
4          sum += vec[i];
5      }
6      return (float) sum / n ;
7  }

```

Ahora podremos hacer nuestro problema original, contar cuántos alumnos tienen una edad mayor a la edad promedio:

```

1  int main() {
2      int n = 0;
3      cout << "Ingrese cantidad de alumnos";
4      cin >> n;
5      int edades[n];
6      cargar(edades, n);
7      float prom = promedio(edades, n);

```

```

8      int cant = 0;
9      for(int i = 0; i < n; i++) {
10         if (edades[i] > prom) {
11             cant++;
12         }
13     }
14     cout << "Hay " << cant << " alumnos mayores que el promedio";
15 }

```

Matriz

En la sección anterior dijimos que los vectores también son llamados arreglos unidimensionales. Tal como vimos en la declaración y acceso a los valores del vector, el vector tiene una única dimensión. Pero existen arreglos de más dimensiones por lo cual los arreglos son clasificados en:

- Unidimensionales (vectores)
- Bidimensionales (tablas o matrices)
- Multidimensionales (tres o más dimensiones)

Una matriz es un arreglo en donde a cada elemento se accede por medio de dos índices.

Declaración

Al igual que con los arreglos unidimensionales, en este tipo de estructura de datos también debemos definir su tamaño antes de utilizarla, con el agregado de la segunda dimensión. De igual forma que antes con los vectores, al declarar la matriz colocaremos entre corchetes las dimensiones, siendo la primera lo que denominaremos como FILAS y la segunda como COLUMNAS.

```

1 int mat[3][4]

```

Esta matriz declarada tendrá 3 filas y cuatro columnas. Pero también es posible que necesitemos que alguna de las dimensiones sea ingresada por el usuario o calculada, para esto podremos utilizar variables al momento de la declaración. Pero al igual que con los vectores estas variables deben tener valor al momento de usarlas para la declaración:

N filas por 4 columnas	3 filas por N columnas	N filas por M columnas
------------------------	------------------------	------------------------

```
int mat[N][4]
```

```
int mat[3][N]
```

```
int mat[N][M]
```

Cabe volver a remarcar que tanto en los vectores como en las matrices los elementos deben ser siempre del mismo tipo. En los ejemplos anteriores, todos los elementos guardados en la matriz serán enteros.

También es posible realizar la inicialización con valores de la matriz de manera similar a los vectores:

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

Acceso a cada elemento

Índice

En el caso de las matrices, como es un arreglo con dos dimensiones, a cada elemento se accede por medio de dos índices, el primero corresponde a la fila y, el segundo, corresponde a la columna.

Veamos un ejemplo para entender el acceso a cada elemento:

```
1 MAT[0][2] = 20  
2 MAT[1][4] = 21  
3 cout << MAT[1][4]
```

1. En la primera acción, asignamos el valor 20 al elemento de la fila 0, columna 2.
2. En la segunda acción asignamos el valor 21 al elemento de la fila 1, columna 4.
3. En la tercera acción mostramos el valor del elemento de la fila 0, columna 2.

¿Qué valor se mostrará por pantalla?

El número 21.

La posición de la fila y la columna se indica por medio de los dos números, teniendo en cuenta que el primero corresponde a la fila y, el segundo, a la columna.

Representación gráfica

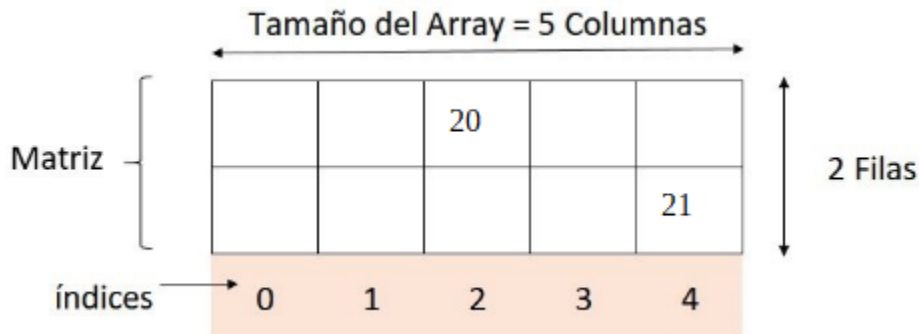
Si tuviésemos la siguiente matriz y asignaciones:

```

1 int mat[2][5];
2 mat[0][2] = 20;
3 mat[1][4] = 21

```

Gráficamente la matriz definida la podemos ver de la siguiente manera:



- MAT sub 0, 2 es igual a 20.
- El valor del elemento de la posición de la fila 0 y columna 2 es 20.

Algoritmos básicos: Cargar Matriz

Veamos el algoritmo que utilizaremos para cargar una matriz con los datos que ingresa un usuario por teclado.

```

1 int main() {
2     int N = 3;
3     int M = 4;
4     int mat[N][M];
5     for(int i = 0; i < N; i++) {
6         for(int j = 0; j < M; j++) {
7             cout << "Ingrese un valor";
8             cin >> mat[i][j];
9         }
10    }
11 }

```

En este ejemplo se cargan todas las posiciones de matriz y el orden de carga es por filas. Es decir que el primer elemento que ingrese el usuario se cargará en la posición fila 0, columna 0. El segundo elemento que ingrese el usuario se cargará en la posición de la fila 0, columna 1. Y así sucesivamente, recorriendo la matriz primero por la fila 0, cuando llegue al último elemento de la fila 0, se pasará a la fila 1 Y así hasta recorrer y cargar toda la matriz.

Si en lugar de cargar por filas deseáramos cargarla por columnas lo único que debemos hacer es generar en el for externo las columnas en lugar de las filas. La forma más clara y sencilla de hacerlo es simplemente intercambiar los for:

```
1  int main() {
2      int N = 3;
3      int M = 4;
4      int mat[N][M];
5      for(int j = 0; j < M; j++) {
6          for(int i = 0; i < N; i++) {
7              cout << "Ingrese un valor";
8              cin >> mat[i][j];
9          }
10     }
11 }
```

De esta forma por cada valor de j que representa una columna se generan todos los valores de i que representan las filas.

Lo más importante a tener en cuenta es siempre que los **índices no excedan el valor de la dimensión** que tienen que iterar, es decir si decimos que i iterara las filas, entonces no puede superar el valor usado al definir la matriz como filas, en este caso N; y lo mismo con la j si la usaremos para las columnas.

Algoritmos básicos: Mostrar Matriz

Del mismo modo que al cargar una matriz se puede recorrer por filas o por columnas, el mostrar los elementos de una matriz también podemos hacer ambos recorridos.

Veamos ahora una rutina que muestra los elementos de una matriz, recorriéndola por columnas. Es decir que el primer elemento que se muestra será el de la fila 0 columna 0, luego el de la fila 1, columna 0, y cuando finalice la columna 0 se pasará a la columna 1. Y así hasta llegar hasta la última columna.

```
1  int main() {
2      int N = 3;
3      int M = 4;
4      int mat[N][M];
5      for(int j = 0; j < M; j++) {
6          for(int i = 0; i < N; i++) {
7              cout << mat[i][j];
8          }
9      }
10 }
```

Matrices como parámetros

Al igual que sucede con los vectores, las matrices sólo pueden pasarse como parámetros por Referencia. Pero al utilizarlas en C/C++ tendremos una limitante más.

Intuitivamente creeríamos que para declarar una matriz como parámetros haríamos algo como:

```
1 void cargarMatriz(int mat[][], int filas, int cols) {
2     for(int i = 0; i < filas; i++) {
3         for(int j = 0; j < cols; j++) {
4             cin >> mat[i][j];
5         }
6     }
7 }
```

Pero el compilador admite que sólo haya una dimensión variable, como cuando lo usamos para vectores, y en este caso tenemos dos dimensiones. Por lo cual deberemos especificar al menos una de las dimensiones en la declaración del procedimiento:

```
1 void cargarMatriz(int mat[][3], int filas) {
2     for(int i = 0; i < filas; i++) {
3         for(int j = 0; j < 3; j++) {
4             cin >> mat[i][j];
5         }
6     }
7 }
```

De esta forma la dimensión variable son las filas y podremos tener N filas, pero siempre serán solamente 3 columnas.

Una forma de paliar esta limitación es usar valores máximos de filas y columnas, de esta forma podremos tener la matriz de N x M siendo $N < \text{MAX_FILAS}$ y $M < \text{MAX_COLS}$ estos valores máximos no pueden ser ni variables ni constantes de las que usamos comunmente sino que deben ser valores definidos para que estén disponibles en tiempo de compilación, esto lo logramos usando la clausula #define:

```
1 #define MAX_FILAS 100
2 #define MAX_COLS 100
3
4 void cargarMatriz(int mat[MAX_FILAS][MAX_COLS], int filas, int cols) {
5     for(int i = 0; i < filas; i++) {
6         for(int j = 0; j < cols; j++) {
7             cin >> mat[i][j];
8         }
9     }
10 }
11 }
```

```

12 int main() {
13     int mat[MAX_FILAS][MAX_COLS]; //Ya defino la matriz con los valores maximos
14     int N, M;
15     cout << "Ingrese cant de filas";
16     cin >> N; //Tomo la cantidad de filas a trabajar, debe ser < MAX_FILAS
17     cout << "Ingrese cant de columnas";
18     cin >> M; // Tomo la cantidad de columnas a trabajar, debe ser < MAX_COLS
19     cargarMatriz(mat, N, M);
20     imprimirMatriz(mat, N, M);
21 }

```

Este algoritmo es más parecido al que habíamos intentado al principio, el único inconveniente será el desperdicio de memoria en los casos en que **N** es mucho menor a **MAX_FILAS** y **M** es mucho menor que **MAX_COLS**.

Veamos un ejemplo completo. Realicemos un programa que permita al usuario cargar N alumnos y por cada alumno debe poder ingresar el legajo, y las tres notas de la materia. Vemos que con un vector no nos alcanzaría para almacenar toda esta información, necesitaríamos al menos 4 vectores: uno para los legajos, y uno más por cada nota. En cambio utilizando matrices podremos definir una matriz de N x 4.

Como sabemos que la cantidad de columnas es fija, podremos usar la forma `mat[N][4]` tanto en la declaración como en las funciones si hicieran falta.

	0 (Legajo)	1 (Nota 1)	2 (Nota 2)	3 (Nota 3)
0	12323	6	3	8
1	22112	4	8	7
2	33223	6	2	9
N	42322	7	2	10

La función `cargarAlumnos` permitirá cargar todos los datos de cada alumno, primero el legajo y luego cada nota. La función podría ser de la siguiente forma:

```

1 void cargarAlumnos(int mat[][4], int cant) {
2     for(int i = 0; i < cant; i++) {
3         cout << "Ingrese el legajo del alumno";
4         cin >> mat[i][0]; //En la primer columna de cada fila estará el legajo
5
6         for(int j = 1; j < 4; j++) {
7             cout << "Ingrese la nota " << j << ": ";
8             cin >> mat[i][j];
9         }
10    }
11 }
12

```

Si quisiéramos imprimir un listado de la siguiente forma:

Alumno:

- Nota 1:
- Nota 2:
- Nota 3:
- Promedio:

El algoritmo podría ser:

```
1 void imprimirListado(int mat[][4], int cant) {
2     int sum = 0;
3     for(int i = 0; i < cant; i++) {
4         cout << "Alumno: " << mat[i][0];
5         sum = 0;
6         for(int j = 1; j < 4; j++) {
7             cout << "    - Nota " << i << ": " << mat[i][j];
8             sum += mat[i][j];
9         }
10        cout << "    - Promedio: " << (float) sum / 3;
11    }
12 }
```

Y el algoritmo principal que llama a nuestros procedimiento:

```
1 int main() {
2     int N;
3     cout << "Ingrese cantidad de alumnos";
4     cin >> N;
5     mat[N][4]; //Solo cuando ya tengo la cantidad de alumnos puedo declarar mat
6
7     cargarAlumnos(mat, N);
8     imprimirListado(mat, N);
9
10 }
```

Estructura de Datos

Volviendo a nuestro ejemplo de edades de alumnos, supongamos que ahora además de pedir la edad de un alumno, también queremos saber el nombre y la cantidad de materias aprobadas. Es decir que **por cada alumno tendremos tres valores** que guardaremos en tres variables distintas. En el caso en que se necesite hacer un procedimiento que imprima los datos de cada alumno, tendrá que tener esos **tres datos como parámetros**.

Hasta ahora no habrás encontrado ninguna dificultad en este problema, pero qué sucede si te decimos que, además de los datos mencionados, también queremos guardar otros valores, como, por ejemplo, el número de documento, el sexo, la cantidad de materias que está cursando.... Tendrás que agregar todos esos datos en la lista de parámetros de cada módulo que

quiera trabajar con los datos de un alumno. Como habrás notado, se estará haciendo más complejo el algoritmo, y hasta cierto punto, difícil para entenderlo y hasta para probarlo.

Evidentemente necesitamos algo para poder guardar de forma más apropiada estos valores. Es en este momento donde necesitamos utilizar las **Estructuras** o Registros de datos.

Veamos cómo definimos un registro para guardar los datos de un alumno en C++:

```
1 struct Alumno {  
2     string Nombre;  
3     int Edad;  
4     int Aprobadas;  
5 };  
6  
7 int main() {  
8     Alumno alum;  
9 }
```

Para definir un Registro debemos definir un nuevo tipo de dato que indique los elementos que contiene y cuáles serán sus nombres.

En este ejemplo estamos definiendo un nuevo tipo de datos que se llama **Alumno** como un Registro. Este Registro tiene tres campos que son: **Nombre**, de tipo de dato String; **Edad**, de tipo de dato Entero y **Aprobadas** que también es de tipo de dato Entero.

Luego definimos una variable que se llama **alum** que es un **Registro del tipo de dato Alumno**. Es decir, que la variable alum guardará tres valores bajo su mismo nombre lógico.

Uno de los principales motivos por el que definimos Registros es para asociar bajo una misma variable a un conjunto de elementos que representan el mismo concepto.

En el ejemplo que te estamos presentando, estos tres valores, el nombre, la edad y la cantidad de materias aprobadas, en su conjunto representan los datos de un alumno, es decir, en otras palabras, que **un alumno está formado por esos tres valores**.

Campos

Seguramente te estarás preguntando que, si bien los tres elementos están asociados a un mismo nombre lógico, debe haber alguna manera de poder acceder a cada dato. Esa manera es utilizando el nombre de cada elemento. **Cada elemento de la Estructura se denomina “campo”**.

En nuestro ejemplo, la estructura **Alumno** tiene tres campos. Y podrás acceder a cada campo por medio del operador punto “.” y el nombre del campo.

```

1  int main() {
2      Alumno alum;
3      cout << "Ingrese la edad del alumno";
4      cin >> alum.Edad; //Se escribe en el campo edad
5
6      cout << alum.Edad; //Se lee el campo edad
7  }

```

- alum -> Nombre de Registro
- "." -> Operador punto
- Edad-> Nombre del Campo

Para acceder a la **Edad** debemos indicarlo con el nombre de la variable alum seguido del operador punto "." y luego el nombre del campo al que queremos acceder, en este caso Edad.

Del mismo modo, si queremos asignar el nombre "Juan" como nombre del alumno debemos hacerlo de siguiente manera:

```
alum.Nombre = "Juan";
```

Algoritmos básicos

Una de las ventajas de utilizar estructuras es que se pueden agrupar en una misma variable todos los datos relacionados lógicamente. De esta manera, podemos manejar la abstracción para el ingreso y la salida de los datos.

Por ejemplo, si estamos trabajando con datos de alumnos, lo más simple sería pensar en una entidad **ALUMNO** como un único conjunto de datos (ocultando los datos que tiene un alumno).

Siguiendo con este concepto, podemos desarrollar los siguientes procedimientos.

```

1  Alumno obtenerAlumno() {
2      Alumno alum;
3      cout << "Ingrese nombre: ";
4      cin >> alum.Nombre;
5
6      cout << "Ingrese edad: ";
7      cin >> alum.Edad;
8
9      cout << "Ingrese materias aprobadas: ";
10     cin >> alum.Aprobadas;
11 }

```

```

12     return alum;
13 }
14
15 void imprimir(Alumno alum) {
16     cout << "Nombre: " << alum.Nombre;
17     cout << "Edad: " << alum.Edad;
18     cout << "Aprobadas: " << alum.Aprobadas;
19 }

```

En la función **obtenerAlumno**, habrás notado que el tipo de retorno es **Alumno**. Esto nos permite simplificar y abstraer el tipo de dato, ya que estamos manejando un alumno como concepto de una entidad (sin importar los datos que tenga).

Por otro lado en el procedimiento imprimir, estamos pasando como parámetro por Valor una variable también del tipo Alumno, con lo cual logramos la misma abstracción.

Al utilizar la estructura y no los datos “sueltos” será mucho más simple el mantenimiento, ya que si querés agregar o eliminar datos al alumno, solo vas a tener que modificar la estructura del registro. Las especificaciones de las funciones y procedimientos que utilizan estas estructuras no se ven modificadas ya que siempre se estará pasando como parámetro un registro de ese tipo.

Diferencias con Arreglos

La principal diferencia que tienen las estructuras de datos o registros respecto de los arreglos es la **posibilidad de almacenar datos de distintos tipos bajo el mismo nombre** de variable. Recordemos que los arreglos, tanto las matrices como **los vectores, siempre son de un único tipo** de datos. Por lo cual si deseamos tener en un mismo lugar el nombre, legajo y las notas de los alumnos, no lo podríamos hacer con un único arreglo.

Podríamos recurrir a una matriz como la que hemos usado anteriormente y además otro vector con los nombres ya que son de otro tipo de dato.

int mat[N][4] y string nombres[N]

Y hacer la correspondencia por números de posición. Lo que daría gráficamente algo como lo siguiente:

	Nombre	0 (Legajo)	1 (Nota 1)	2 (Nota 2)	3 (Nota 3)
0	Maria	0 12323	6	3	8
1	Pepe	1 22112	4	8	7
2	Julia	2 33223	6	2	9
N	Carlos	N 42322	7	2	10

Lo cual nos obligaría a estar accediendo y manteniendo ambas estructuras por separado, lo cual no es una abstracción muy potente.

En cambio con las estructuras generamos nuevos tipos de datos, que pueden combinar cualquier otro tipo de dato, los primitivos, los arreglos, y los nuevos que nosotros creemos. Por lo cual podemos realizar un vector de estructura del siguiente tipo:

Alumno alumnos[N]

Y entonces nuestras funciones de carga e impresión de datos ya no trabajan con datos “sueltos” sino con estructuras que representan abstracciones mejor pensadas:

```
1 void cargarAlumnos(Alumno alumnos[], int cant) {
2     for(int i = 0; i < cant; i++) {
3         alumnos[i] = obtenerAlumno();
4     }
5 }
6
7 void imprimir(Alumno alumnos[], int cant) {
8     for(int i = 0; i < cant; i++) {
9         imprimirAlumno(alumnos[i]);
10    }
11 }
```

Además de esta forma es bastante más específico el dato que se guarda en cada lugar, y no por posiciones, como lo hacemos en las matrices o vectores.

Una posible estructura para poder guardar todos los datos del alumno y sus notas sería:

```
1 struct Alumno {
2     int Legajo;
3     string Nombre;
4     int Nota1;
5     int Nota2;
6     int Nota3;
7 };
```

Y podemos reescribir nuestros algoritmos para que hagan uso de esta estructura en lugar de una matriz de enteros.



Bibliografía utilizada

Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.

- Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)
- Debugging with CodeBlocks:
http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks